

ARBEITSBERICHTE DES INSTITUTS FÜR MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG (INFORMATIK)

FRIEDRICH ALEXANDER UNIVERSITÄT ERLANGEN NÜRNBERG

HERAUSGEBER : H. Billing, W. Händler, U. Herzog,
F. Hofmann, K. Leeb, P. Mertens, H. Müller, G. Nees,
H. Niemann, D. Seitzer, H. J. Schneider, H. Wedekind

AUFTRAGSSYSTEM FÜR EINE MEHRPROZESSORANLAGE

Franz X. Wurm

DISSERTATION

BAND 13 · NUMMER 7 · ERLANGEN · OKTOBER 1980

1980 424



ISSN 0344 - 3515



Digitized by the Internet Archive
in 2026

ARBEITSBERICHTE DES INSTITUTS FÜR MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG (INFORMATIK)

FRIEDRICH ALEXANDER UNIVERSITÄT ERLANGEN NÜRNBERG

HERAUSGEBER : H. Billing, W. Händler, U. Herzog,
F. Hofmann, K. Leeb, P. Mertens, H. Müller, G. Nees,
H. Niemann, D. Seitzer, H. J. Schneider, H. Wedekind

AUFTRAGSSYSTEM FÜR EINE MEHRPROZESSORANLAGE

Der Technischen Fakultät
der Universität Erlangen-Nürnberg
zur Erlangung des Grades
DOKTOR - INGENIEUR

vorgelegt von
Franz X. Wurm

Erlangen 1980

DISSERTATION

BAND 13 · NUMMER 7 · ERLANGEN · OKTOBER 1980



Als Dissertation genehmigt
von der Technischen Fakultät
der Universität Erlangen-Nürnberg

Tag der Anmeldung : 28. April 1980

Tag der Promotion : 6. August 1980

Dekan : Prof. Dr. H. Niemann

Berichterstatter : Prof. Dr. F. Hofmann

Prof. Dr. W. Händler

ISSN 0344-3515

Zusammenfassung:

In dieser Arbeit wird ein benutzerfreundliches und effizient zu implementierendes Auftragssystem für eine Mehrprozessoranlage beschrieben.

Das Auftragssystem ist geeignet für symmetrische wie auch asymmetrische und hierarchische Systeme. Es läßt auch zu, daß auf den einzelnen beteiligten Prozessoren weitgehend autonome und lokal unterschiedliche Betriebssysteme installiert sein können.

Der Vorschlag löst alle notwendigen Auftragsabläufe und damit verbundene Synchronisationsprobleme. Besondere Beachtung wurde der dynamischen Anpassung und Erweiterung von Dienstleistungen und der Diagnose und Leistungsbewertung gewidmet.

Die Arbeit beinhaltet neben Einführung und Herleitung der Anforderungen eine formale Spezifikation des Auftragssystems in Form eines abstrakten Datentyps und einen Implementierungsvorschlag.

Lösungsvorschläge für wichtige bekannte Probleme zeigen die Leistungsfähigkeit des Systems.

Inhaltsverzeichnis

0. Einleitung und Motivation	1
1. Das verwendete Mehrprozessormodell	3
1.1 Klassifizierung von Mehrprozessorsystemen	3
1.2 Bewertung einiger Eigenschaften	5
1.3 Einsatzgebiete für das geplante System	6
1.4 Entwurfsidee	7
1.5 Schutzaspekte	13
1.6 Anschluß peripherer Geräte	14
1.7 Beschreibung des Mehrprozessormodells	14
2. Betriebssysteme für Mehrprozessoranlagen	17
2.1 Vergleich Einprozessor-Mehrprozessoranlagen	17
2.2 Komplexität von Mehrprozessorbetriebssystemen	20
3. Auftragssystem für eine Mehrprozessoranlage	23
3.1 Problemstellung	23
3.2 Mögliche Lösungswege	23
3.3 Forderungen an eine Lösung	24
3.4 Lösungsidee	26
3.4.1 Überblick	26
3.4.2 Ablauf eines Auftrages	29
3.4.3 Diagnose und Bewertungsfunktionen	30
3.5 Datenobjekte	30
3.6 Funktionsumfang	34
3.6.1 Erzeugen und Löschen der notwendigen Datenstrukturen	36
3.6.2 Auf- und Abbau von Verbindungen	38
3.6.3 Senden und Empfangen von Nachrichten	40
3.6.4 Synchronisationsfunktionen	41
3.6.5 Diagnose und Bewertung	42
3.7 Realisierung des Objektschutzes	42
3.8 Nachteile und Grenzen des vorgestellten Systems	45

4. Formale Spezifikation der einzelnen Funktionen	46
4.1 Spezifikationsmethode	46
4.2 Datenstrukturen	48
4.3 Exportierte Funktionen	51
4.3.1 INITPORTLIST	51
4.3.2 CREATEPORT	52
4.3.3 DELETEPORT	53
4.3.4 GETNEWPORTNAME	54
4.3.5 CREATELUAREA	55
4.3.6 EXPANDLUAREA	56
4.3.7 SETLUATTR	57
4.3.8 DELETELUAREA	58
4.3.9 ASSIGN	59
4.3.10 UNASSIGN	60
4.3.11 LINK	61
4.3.12 UNLINK	62
4.3.13 PRIUNASSIGN	63
4.3.14 PRIUNLINK	64
4.3.15 DEPOSIT	65
4.3.16 CANCEL	66
4.3.17 ACCEPT	67
4.3.18 REJECT	68
4.3.19 REPLY	69
4.3.20 TESTLU	70
4.3.21 ORLU	71
4.3.22 UNORLU	72
4.3.23 WAITLU	73
4.3.24 CHANGEACC	74
4.4 Fehlerliste	75

5. Implementierungsvorschlag und Detailspezifikation	76
5.1 Vorbemerkungen	76
5.2 Synchronisationsprobleme	77
5.3 Globale und lokale Funktionen	79
5.4 Datenbeschreibung	80
5.4.1 PORTLIST	80
5.4.2 LUAREA	82
5.4.3 Parameterblock	83
5.5 Detailspezifikation der wichtigsten Funktionen	84
5.5.1 CREATEPORT	84
5.5.2 DELETEPORT	86
5.5.3 ASSIGN	87
5.5.4 UNASSIGN	89
5.5.5 DEPOSIT	90
5.5.6 REPLY	91
5.4.7 WAITLU	92
6. Behandlung ausgewählter Problemstellungen	93
6.1 Bedienprozesse	93
6.2 Ein/Ausgabe	94
6.3 Allgemeine Dienstleistungen und Makro-Pipelining	96
6.4 Nachrichtenaustausch	98
6.5 Synchronisation	99
Schlußbemerkung	100
Literaturverzeichnis	101

0. Einleitung und Motivation

Mehrprozessoranlagen und Rechnerverbundsysteme gewinnen immer mehr an Bedeutung. Ihre Entwicklung ging einerseits den Weg, das zentrale Prozessorelement zu vervielfachen, um für spezielle Anwendungen Berechnungsschritte parallel ausführen zu können, und führte zu den sog. Feldrechnern. Auf der anderen Seite wurden größere Anlagen durch spezialisierte Prozessoren für bestimmte Aufgaben erweitert, angefangen bei einfachen Kanalwerken bis zu eigenständigen Peripherierechnern, Bedienungsrechnern, Rechnern für Datenfernübertragung usw. Diese Entwicklung wurde durch die Innovation der Technologie stark beschleunigt. So bieten Mikroprozessorhersteller häufig schon Bausteine an, die den Aufbau von Mehrprozessoranlagen unterstützen, und der Anteil der Prozessorkosten an den Gesamtkosten wird immer geringer.

Sieht man von den speziellen Feldrechnern ab, so erscheinen Systeme mit unterschiedlichen Prozessorelementen durchaus sinnvoll. Um die Komplexität des Betriebssystems in einem solchen System in einer noch handhabbaren Größe zu halten, muß es gut strukturiert sein und für die einzelnen Rechner möglichst unabhängige Teilsysteme enthalten.

Innerhalb der gemeinsamen Teile des Betriebssystems kommt der Auftragvergabe zwischen den einzelnen Rechnern eine zentrale Bedeutung zu. Das Auftragssystem und die von ihm benötigten Typen stellen den eigentlichen Kern des Betriebssystems für eine solche Mehrprozessoranlage dar.

Die vorliegende Arbeit versucht ein flexibles, benutzerfreundliches und nicht zu aufwendig zu implementierendes Auftrags-system für eine Mehrprozessoranlage vorzustellen. Dabei werden in Kap.1 die zugrundegelegten Anwendungsgebiete aufgeführt und ein idealisiertes Rechnermodell vorgestellt.

Nach einigen für das System wichtigen Bemerkungen über Betriebssystemprobleme bei Mehrprozessoranlagen in Kap.2 werden in Kap.3 die Grundideen und einige wichtige Aspekte beschrieben. Kap.4 bringt eine vollständige Spezifikation des Auftragssystems in Form eines abstrakten Datentyps. Kap.5 beschreibt allgemeine Implementierungsentscheidungen und bringt für die wichtigsten Funktionen einen Implementierungsvorschlag in Form eines "Grobablaufes". Hierbei wurden besonders die auftretenden Synchronisationsprobleme behandelt.

Kap.6 schließlich zeigt anhand der wichtigsten bekannten Auftragsklassen die Leistungsfähigkeit des Systems.

1. Das verwendete Mehrprozessormodell

1.1 Klassifizierung von Mehrprozessoranlagen

Rechenanlagen lassen sich nach unterschiedlichen Merkmalen klassifizieren. So wird in /StoH75/ der Grad der Parallelität von Instruktionsströmen und Datenströmen als Merkmal verwendet; danach erhält man folgende Klassen:

SISD (Single Instruction Single Data)

SIMD (Single Instruction Multiple Data)

MISD (Multiple Instruction Single Data)

MIMD (Multiple Instruction Multiple Data)

Vom Betriebssystemstandpunkt aus interessiert in erster Linie die Anzahl möglicher paralleler Instruktionsströme, da sie angeben, wieviele Prozesse (Prozeßbegriff etwa nach /KerS74/ oder /KerS74a/) parallel laufen können. Man kann bei einer SIMD-Anlage die verarbeiteten Daten als ein einziges Datum "Vektor" oder "Feld" betrachten und die Anlage als Spezialfall eines SISD-Systems betrachten. Ebenso kann man MISD, in der oben zitierten Arbeit nur der Vollständigkeit wegen aufgeführt, als speziellen Arbeitsmodus eines MIMD-Systems auffassen.

Grundsätzlich muß hier erwähnt werden, daß sich je nach Betrachtungsstandpunkt jede Anlage als SISD oder MIMD einordnen läßt. Auf der Logik- und Mikroprogrammirebene wird es in jedem Rechner parallele Aktionen geben, die auf unterschiedliche Daten zugreifen oder bitparallel Verknüpfungen durchführen.

Andrerseits kann vom Benutzerstandpunkt die Bearbeitung einer Datei durch ein paralleles Programmsystem als ein Bearbeitungsschritt aufgefaßt werden.

Bei den weiteren Ausführungen soll deshalb als Prozessor (PU) eine Einheit verstanden werden, die einen Befehlsstrom abarbeiten kann, mit Befehlen, die in ihrer Mächtigkeit zwischen den Befehlen von üblichen Minirechnern und denen von Algorithmischen

Sprachen, wie etwa Pascal, Fortran, Algol und - um Fehldbefehle einzuschließen - etwa auch APL liegen.

Als Ein- oder Mehrprozessoranlagen werden dann Anlagen bezeichnet, die eine oder mehrere solche PUs enthalten.

So definierte Mehrprozessoranlagen lassen sich jetzt weiter klassifizieren, z.B. nach der Stärke und Struktur ihrer Kopplungen. Man kann unterscheiden zwischen Mehrprozessoranlagen, bei denen alle PUs zusammen einen gemeinsamen Speicher besitzen, und Rechnernetzen, bei denen jeder Prozessor seinen eigenen Speicher besitzt und mit anderen nur über, als Ein-Ausgabe Geräte betriebene, Nachrichtenwege Information austauschen kann. Dazwischen sind beliebige Varianten mit mehr oder minder regulären Vermaschungen möglich. Es soll hier schon erwähnt werden, daß die vorliegende Arbeit sich auf Rechenanlagen bezieht, bei denen die einzelnen Prozessoren eigene Speicher (private Speicher) besitzen und auch über von mehreren PUs adressierbare Speicher (gemeinsame Speicher) Informationen austauschen können.

Neben der Kopplungsstärke werden bei der Klassifizierung von Mehrprozessoranlagen auch Begriffe wie symmetrisch/asymmetrisch und hierarchisch verwendet.

Symmetrisch heißt, daß die einzelnen Prozessoren gleichartig sind und gleiche Befehlsströme abarbeiten können.

Hierarchisch heißt u.a., daß nicht jeder Prozessor alle Betriebssystemfunktionen durchführen kann.

Nach /WitF75/ werden größere Datenverarbeitungsanlagen der Zukunft verschiedene Prozessoren mit unterschiedlicher Leistungsfähigkeit besitzen, die den einzelnen Aufgabenklassen wie Ausführung rechenintensiver Prozesse, Steuerung der Peripherie, Dateiverwaltung, Auftragsverwaltung, Kommunikation usw. angepaßt sind. In einem solchen System wird nicht jeder Rechner alle Betriebssystemobjekte "kennen" und zugehörige Funktionen ausführen können, d.h. man wird ein asymmetrisch und zu einem gewissen Grad hierarchisches System haben.

1.2 Bewertung einiger Eigenschaften

Es gibt zwei Wege, zu einer Mehrprozessoranlage zu kommen. Man nimmt eine Einprozessoranlage und versucht die Leistung zu erhöhen, indem man an den zentralen Speicher zusätzliche Prozessoren (PUs) anschließt. Oder man hat schon mehrere getrennte Einprozessoranlagen und schließt diese über Rechnerkoppelleinrichtungen zu einem Netz zusammen.

Im ersten Fall kann man durch meist nicht zu umfangreiche Erweiterungen am Einprozessorsystem ein Betriebssystem für eine solche Anlage erstellen. Die Komplexität wird dabei nicht wesentlich höher als bei der Einprozessoranlage sein. Einen Nachteil bilden aber die Zugriffskonflikte am gemeinsamen Speicher. Abhängig von Verteilung und Ausführungszeiten einzelner Befehle, in Relation zur Zykluszeit des Speichers, werden bei Speicherzugriffen gegenseitige Behinderungen der Prozessoren auftreten. Auch wenn man diese Konflikte durch entsprechende Strukturierung der Speicher (Blockbildung, Cachememory usw.) mindern kann, erreicht man ab einer gewissen Anzahl von Prozessoren keine zusätzliche Leistungssteigerung /GerB76/.

Weiter hat die Erfahrung gezeigt (eine von Amdahl für IBM-Anlagen erstellte Faustregel), daß in einem "ausgewogenen" System die Speichergröße in Megabyte in der gleichen Größenordnung liegen soll wie die Anzahl MIPS (Mega Instructions per second). Wenn also bei einem Mehrprozessorsystem die Leistung auf mehrere MIPS angehoben wird, muß auch der gemeinsame Speicher entsprechend groß werden. Das hat zur Folge, daß auch die Adressierungsmechanismen der einzelnen Prozessoren erweitert werden müssen, obwohl u.U. die benutzten gemeinsamen Bereiche viel kleiner sind.

Im praktischen Einsatz befindliche Systeme des genannten Typs haben meist auch nur 2-4 Prozessoren (z.B. TR-Dreikernanlage des RRZE).

Die reinen Rechnernetze am anderen Ende der Skala besitzen keine gemeinsamen Speicher. Jeder Informationsaustausch muß über, als Ein-Ausgabegeräte betriebene Kopplungsbausteine und ein Nachricht-

tennetz, durchgeführt werden. Dabei treten folgende Probleme auf:

- Man muß sich bezüglich der Übertragungsprotokolle, Datenformate und Kommandos einigen. (Ein Problem, das nicht allein auf diesen Typ beschränkt ist.)
- Der Informationsaustausch wird aufwendig, wenn viele kurze Informationen übertragen werden müssen.
- Es ist schwierig, in einem solchen System den jeweiligen Gesamtzustand festzustellen und zu überwachen (z.B. für Diagnose oder Tuning)

Wieweit die genannten Probleme in Erscheinung treten, hängt natürlich auch hier vom Einsatzgebiet ab.

1.3 Einsatzgebiete für das geplante System

Die Beurteilungskriterien eines Systems hängen stark vom jeweiligen Einsatzgebiet ab. Da man aber meist nicht alle Anwendungsgebiete im voraus kennt oder den Aufwand auf eine größere Zahl von Anwendungen verteilen will, versucht man, möglichst allgemein zu bleiben. Die Existenz von Spezialrechnern wie dem klassischen Staran /RudJ72/ oder neuere Systeme wie Pluribus /OrnS75/, Siemens SMS /KobR76/ und vielen anderen, lassen Zweifel entstehen, ob man ein "general purpose system" für alle Anwendungen optimal auslegen kann.

Wie bei vielen anderen komplexeren technischen Systemen, wird man deshalb auch bei Computern mit einer größeren Anzahl unterschiedlicher Endprodukte rechnen müssen. Selbstverständlich versucht man, diese Endprodukte - soweit möglich - aus gemeinsamen Halbprodukten und nach gemeinsamen Konstruktionsprinzipien aufzubauen und Systemfamilien oder Sippen zu bilden /HabA76/.

Bei dem in der vorliegenden Arbeit beschriebenen Auftragssystem handelt es sich um einen Baustein (Modul), der möglichst universell einsetzbar sein soll. Aber auch hier hat sich gezeigt, daß

man bei vielen Entscheidungen nur für bestimmte Einsatzgebiete optimieren kann. Der Arbeit sind deshalb die folgenden Einsatzgebiete zugrunde gelegt. Das verwendete System soll dort eingesetzt werden können, wo auch der Einsatz einer Einprozessoranlage noch möglich wäre und zur Zeit auch häufig verwendet wird, und wo die benötigten Rechenleistungen räumlich (oder wenigstens logisch) getrennt anfallen, aber doch zu einem Gesamtsystem zusammengefaßt werden kann. Man kann es als distributed processing aber nicht unbedingt als distributed system bezeichnen. Beispiele hierfür wären ein Prozeßsteuerungssystem, bei dem zur Reduzierung des Datentransports, Erhöhung der Reaktionsgeschwindigkeit und u.U. auch der Ausfallsicherheit, ein Teil der Computerleistung vor Ort durch eigene Rechner erbracht wird. Ein weiteres Beispiel wäre ein System mit intelligenten Terminals oder auch nur die Steuerung der Peripherie durch eigene Rechner.

1.4 Entwurfsidee

Die in 1.3 genannten Einsatzgebiete und die in 1.2 aufgeführten Nachteile der beiden extremen Vertreter von Systemen mit mehreren Prozessoren oder Rechnern führen zwangsläufig zu folgendem Rechnermodell:

Das System besteht aus einzelnen Prozessorelementen (PE), die private Speicher besitzen und die über eine Menge von gemeinsamen Speichern verknüpft sind, auf die jeweils ein Teil der PEs zugreifen kann. Es wird nicht vorausgesetzt, daß alle PEs auf alle gemeinsamen Speicher zugreifen können, noch soll die Struktur regulär sein.

Die einzelnen PEs brauchen nicht gleich zu sein, sondern können wie schon in 1.1 erwähnt, den einzelnen unterschiedlichen Aufgabengebieten optimal angepaßt sein (rechenintensive Aufgaben, Steuerung peripherer Geräte, Textverarbeitung, interne Verwaltungsaufgaben, Diagnose usw.).

Eine ideale Lösung wäre die in Bild 1.1 angegebene Struktur, bei der über ein Verknüpfungswerk beliebige Verbindungen zwischen Prozessoren und Speichern schaltbar sind.

Da durch die privaten Speicher ein hoher Anteil von Speicherzugriffen lokal abgesättigt wird, und nicht jeder Prozessor auf alle gemeinsamen Speicher gleichzeitig zugreift, ist ein Verbindungswerk mit wesentlich niedrigerer Konnektivität ausreichend, als es z.B. im bekannten C.mmp-System /WulW72/ verwendet wird.

Die Verbindungen sollen dabei aus noch zu erwähnenden Gründen von einem zentralen Prozessor oder Prozessorsystem kontrolliert werden. Die einzelnen PEs dieses Systems können selbst auch wieder Mehrprozessorsysteme sein.

Als aus der Literatur bekanntes Beispiel in Richtung der beschriebenen Architektur kann CM* /SwanR77/ von der Carnegie Mellon University angeführt werden. Eine einfachere Lösung des Vorschlages stellt auch die in Bild 1.2 gezeigte Struktur dar, bei der Prozessoren mit lokalen Speichern über einen gemeinsamen Bus und einer zentralen Zugriffskontrolle auf Teile der "lokalen" Speicher der anderen Prozessoren zugreifen können. Weitere Varianten wären das am Institut vorhandene System aus 3 Interdata-Rechnern mit je einem lokalen Speicher und einem zusätzlichen gemeinsamen Speicher (Bild 1.3) und das System EGPA /HänW75, HänW77/, bei dem wechselseitige Zugriffe nur auf Speicher innerhalb einer "beschränkten" Nachbarschaft möglich sind (Bild 1.4).

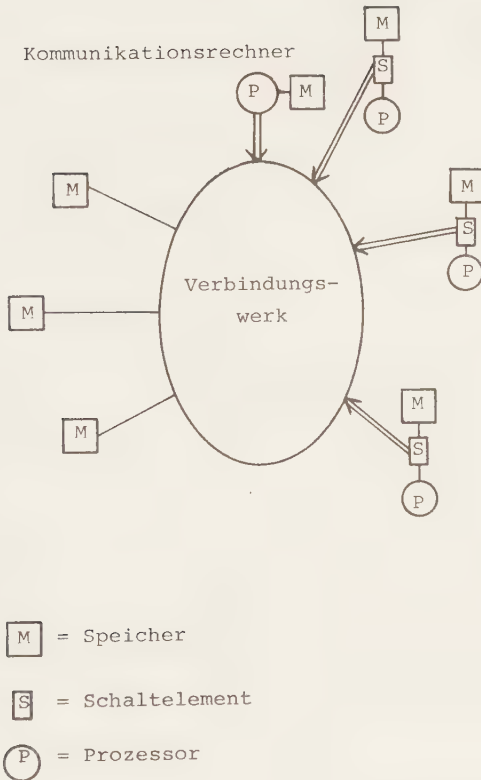
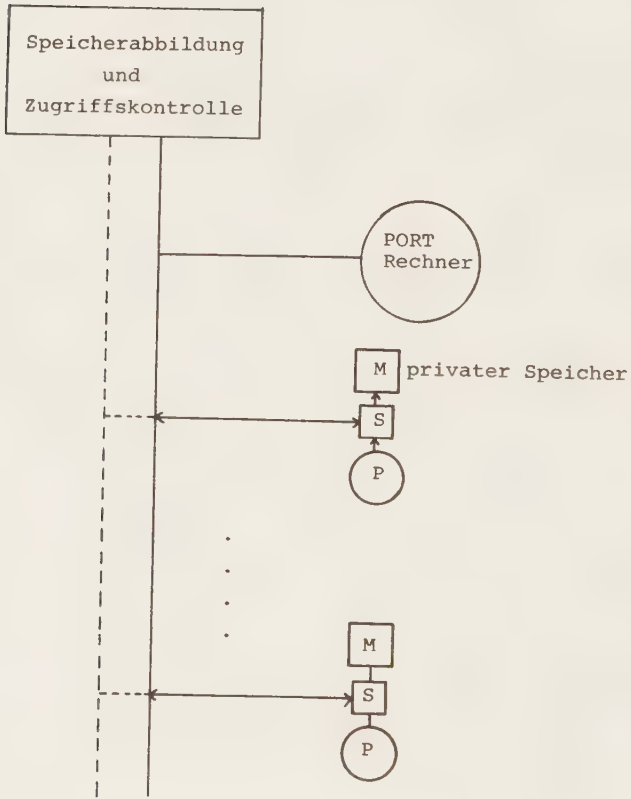
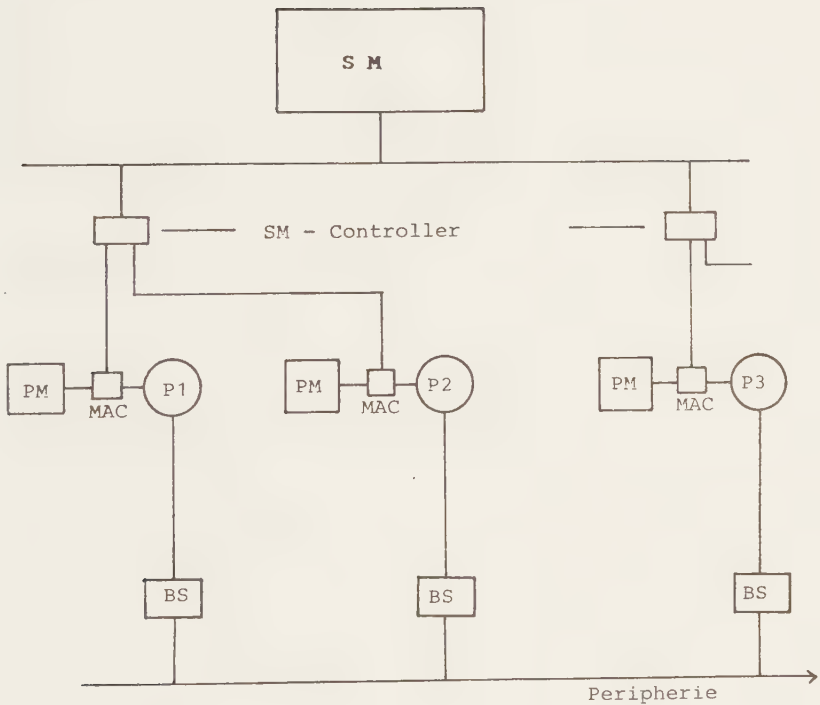


Bild 1.1: Konzept einer Mehrprozessoranlage



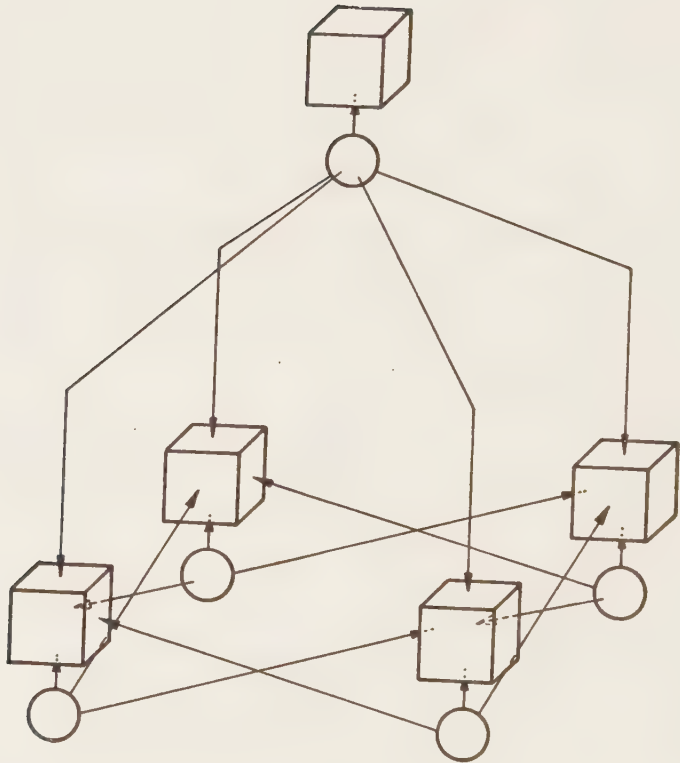
(Mehr-) BUS-System

Bild 1.2: Einfaches Beispiel einer Mehrprozessoranlage



SM = Shared Memory
 PM = Private Memory
 MAC= Memory Access Controller
 BS = Bus Switch
 Pi = Interdata 7/32

Bild 1.3 SYMPOS - Konfiguration aus Interdata-Rechnern



○ = Prozessor

□ = Speicher

→ = Verbindung (Speicherzugriffsmöglichkeit)

Bild 1.4: Grundelement der EGPA-Pyramide

1.5 Schutzaspekte

Um in einem System die Wirkung von Programmfehlern zu beschränken oder Informationsschutz durchführen zu können, ist es notwendig, einem laufenden Prozeß nur auf bestimmte Objekte Zugriff zu erlauben /JonA77/. Dies bedeutet auf der untersten Ebene, daß Zugriffe auf den Hauptspeicher über Hardwareeinrichtungen kontrolliert werden müssen. In einer Einprozessor- oder stark zentralisierten Mehrprozessoranlage werden diese Schutzseinrichtungen von einem sog. Sicherheitskern verwaltet, der auf dem oder den Prozessoren läuft. Geht man dagegen über zu einem System, wie es in 1.2 beschrieben ist, mit unterschiedlichen Prozessoren und wie in 2. gezeigt werden wird, auch unterschiedlichen Betriebssystemkomponenten, und beachtet man weiter, daß in diesem System PE's anschaltbar oder tauschbar sein sollen, so muß man die Problematik der Fehlereinschränkung beim Multiprogramming entsprechend auf das Multiprocessing übertragen. D.h., man kann nicht mehr davon ausgehen, daß die Prozessoren fehlerfreie Hardware und Sicherheitskerne besitzen.

In einem solchen System wird man deshalb in den einzelnen PE's weiterhin lokale Schutzseinrichtungen und Sicherheitskerne benötigen, die zunächst nur die lokalen Zugriffe kontrollieren. Innerhalb des Gesamtsystems braucht man aber zusätzliche Mechanismen, die den Zugriff zwischen einzelnen PE's und gemeinsamen Speichern kontrollieren, wobei PE's hier als jeweils ein einziges Subjekt betrachtet werden. Im gewünschten System werden deshalb eine oder mehrere PE's die Funktion von Verknüpfungs- oder Kommunikationsrechnern erhalten, die die Zugriffscapabilities auf gemeinsame Speicher verwalten und auf Anforderung für die restlichen PE's entsprechende "Verbindungen" schalten. Daß an diese Elemente erhöhte Zuverlässigkeitsansprüche gestellt werden, braucht nicht weiter diskutiert zu werden. Eine hardwaremäßige Realisierung dieses Zugriffsschutzes kann man sich in einfacher Weise so vorstellen, daß durch die Anschaltungshardware an das Verbindungsnetz oder den gemeinsamen Bus eine, vom einzelnen PE nicht manipulierbare, Identifikation bei jedem Transfer mitgeliefert und dadurch (fest oder assoziativ adressiert) ein zugehöriger Satz von Zugriffsschutzregistern angesteuert wird.

1.6 Der Anschluß peripherer Geräte

Die Entwicklung bei peripheren Geräten ging den Weg von relativ unintelligenten Geräten, bei denen sehr viel gerätespezifischer Steueraufwand vom Hauptrechner durchgeführt werden mußte, in Richtung zu Geräten mit intelligenten Steuereinheiten, die immer mehr unabhängig und parallel zur Zentraleinheit arbeiten können. Während in den zurückliegenden Jahren versucht wurde, die Schnittstelle zu peripheren Geräten auf der physikalischen Ebene zu standardisieren, so daß eine Gerätesteuerung physikalische Blöcke selbständig aus- und eintransferieren konnte, erscheint es durch den fortschreitenden Einsatz von Mikroprozessoren durchführbar und sinnvoll, die Schnittstelle zwischen "rechnenden" und "peripheren" Elementen auf eine Ebene anzuheben, die in Richtung dessen geht, was bei bestehenden Systemen als logische E/A-Ebene bezeichnet wird. Funktionen, wie mehrfache Leseversuche bei fehlerhaften Zugriffen auf magnetische Datenträger, Verwaltung des Speicherplatzes auf einer Magnetplatte, oder Blockung, Entblockung und Pufferung von Datensätzen auf Magnetbändern, können vom "Geräterechner" selbst durchgeführt werden.

Eine konsequente Weiterführung dieser Gedanken führt dazu, daß periphere Geräte, einzeln oder zu Gruppen zusammengefaßt, einen eigenen Steuerrechner mit privaten Speichern besitzen und als gleichberechtigter Modul an das Verknüpfungswerk der Mehrprozessoranlage angeschlossen sind. Die Kommunikation zwischen Anwenderprozeß und diesen Ein-Ausgabeprozessen ist eine der wesentlichen Funktionen des im folgenden beschriebenen Auftragssystem.

1.7 Beschreibung des Mehrprozessormodells

Faßt man die bisherigen Gedanken zusammen, so kommt man zu einem System, wie es in Abbildung 1.3 dargestellt ist. Der einzelne Baustein besteht jetzt aus einem Prozessor, einem lokalen Speicher, einer (auch leeren) Menge von peripheren Geräten, und einer Anschaltung an das Verknüpfungsnetz. Nur diese Anschaltung muß bei allen Bausteinen funktionsgleich sein. Die einzelnen Moduln sind

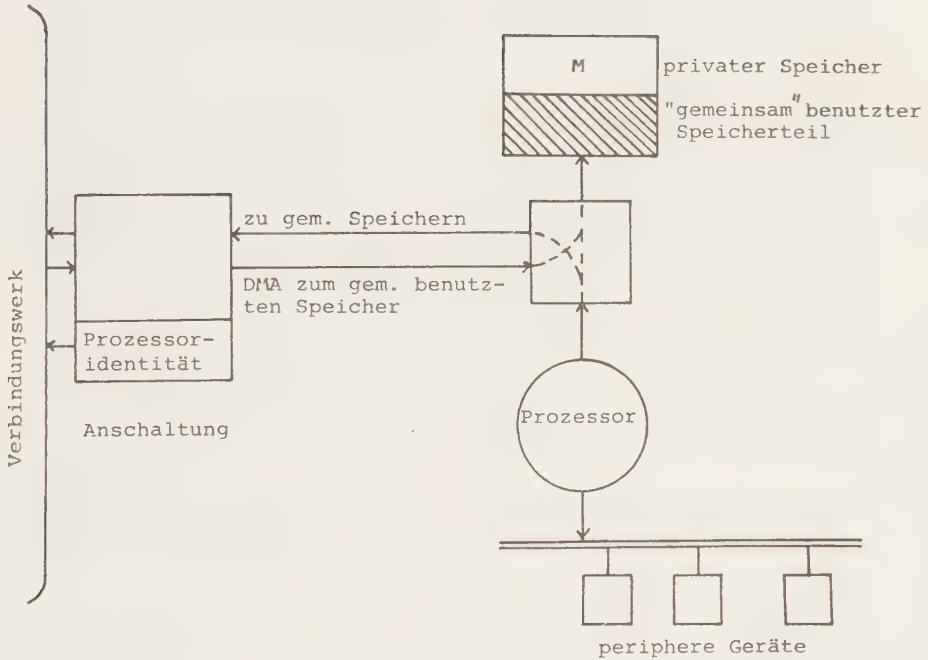


Bild 1.3: Rechnermodul innerhalb der Mehrprozessoranlage

über ein Verknüpfungswerk, das selbst Prozessoren und Speicherelemente enthält, verbunden. Dieses Verknüpfungswerk soll als Kommunikationsrechner und später als PORT-Rechner bezeichnet werden.

Auf die Speicherelemente innerhalb des Verknüpfungswerkes darf wegen der in 1.3.3 aufgeführten Sicherheitsaspekte nur der Kommunikationsrechner selbst Zugriff haben.

Die in 1.3.2 geforderten gemeinsamen Speicher sind in diesem System bei einzelnen Moduln realisiert, d.h. Teile des zunächst lokalen Speichers eines Moduls werden anderen Prozessoren über das Verbindungswerk zugänglich gemacht. Dies erscheint sinnvoll, wenn man bedenkt, daß im allgemeinen ein Prozeß als "Owner" für ein Objekt verantwortlich ist und im Bedarfsfall anderen Prozessen (auf fremden Prozessoren) Zugriffscapabilities überträgt für Dienstleistungen auf diese Objekte.

Der vorliegende Entwurf schließt aber die Existenz von unabhängigen gemeinsamen Speichern nicht aus. Wegen der unterschiedlichen Leistungsfähigkeit und Zuverlässigkeit (Vertrauenswürdigkeit) der einzelnen Moduln kann der für die Einhaltung der Schutzaspekte verantwortliche Sicherheitskern nicht im "distributed processing" auf allen Moduln laufen, sondern soweit dies gemeinsame Objekte betrifft, nur auf dem Kommunikationsrechner.

Der Kommunikationsrechner hat deshalb Zugriff auf alle gemeinsam betreibbaren Speicherbereiche. Alle Aufträge zwischen Prozessoren laufen über den Kommunikationsrechner, der dem Auftragnehmer für die Dauer der Durchführung Zugriff auf ein Objekt im "gemeinsamen" Speicher ermöglicht und bei Auftragsende die Zugriffscapability wieder entzieht.

2. Betriebssysteme für Mehrprozessoranlagen

2.1 Vergleich von Ein- und Mehrprozessoranlagen

Betrachtet man die Anwenderschnittstellen von Ein- und Mehrprozessorbetriebssystemen, so kann zunächst festgestellt werden, daß alle Funktionen, die ein Einprozessorsystem anbietet, auch bei der Mehrprozessoranlage nötig sind.

Wieweit zusätzlich Funktionen vorhanden und notwendig sind, hängt stark vom jeweiligen System und Einsatzbereich ab. Besitzt z.B. ein Rechenzentrumsrechner zur Erhöhung der Rechenleistung und Verfügbarkeit mehrere gleichartige Prozessoren, so sollte der eigentliche Anwender davon nichts merken. Denkt man dagegen an verteilte Prozeßrechnersysteme oder Systeme mit unterschiedlichen Prozessoren, so erscheinen Funktionen zur Zuordnung Prozeß-Prozessor (Prozessoraffinität) notwendig und wünschenswert und/oder zumindest unumgänglich dann, wenn eine automatische Plazierung zu aufwendig wird. Solange man aber zusätzliche Funktionen nicht verwenden will, sollten Programme ohne Änderungen von der Einz zur Mehrprozessoranlage transportiert werden können, d.h. die Anwenderschnittstellen werden sich nicht wesentlich ändern.

Auch bei den internen Funktionen gilt zunächst, daß alle Probleme des Einprozessorsystems auch bei der Mehrprozessoranlage vorhanden sind und analog gelöst werden können. Die Gewichtung der Funktionen und Implementierungsentscheidungen können sich ändern, z.B. kann die Entscheidung, ob ein Systemdienst als Prozedur oder als eigener Prozeß abläuft, unterschiedlich sein. Auch wird man bei Mehrprozessorsystemen der Optimierung des Betriebsmittels Prozessor keine so große Bedeutung zumessen wie bei Einprozessoranlagen, d.h. keine zu aufwendigen Vergabealgorithmen und keine Scheu vor höherem dynamischen Aufwand, wenn es die Modularität und Sicherheit erhöht.

Zusätzlich zum Einprozessorsystem hat man bei der Mehrprozessoranlage folgende Probleme zu lösen:

(Die Aufstellung erhebt nicht den Anspruch, vollständig zu sein)

- Verwaltung der einzelnen Prozessoren
(Zustand: auf ihm laufender Prozeß...)
- Verwaltung der unterschiedlichen Speicherbereiche
- Prozessoraffinität
(Bestimmte Prozesse sind nur auf Teilmengen von Prozessoren lauffähig, u.U. mit unterschiedlichen Prioritäten)
- Bei Interrupt (z.B. Geräteendmeldung, Preemption) muß entschieden werden, welcher Prozessor unterbrochen wird.
- Realisierung sog. kritischer Abschnitte mit Hilfe von LOCK-Variablen und Spezialbefehlen wie "Test and Set".
- Erweiterte Testfunktionen
(parallele Abläufe sind schwieriger zu verfolgen als sequentielle)
- Auftrags-, Datenaustausch und Synchronisation zwischen Prozessoren
- Vorkehrungen, die beim Ausfall einzelner Prozessoren ein Weiterarbeiten ermöglichen, um nicht statt der oft zitierten Erhöhung der Ausfallsicherheit eine Verschlechterung zu erhalten (dadurch, daß der ausgefallene Prozessor wichtige Verwaltungsdaten gesperrt hält).

Der letzte Punkt soll folgendes Problem ansprechen. Bei Einprozessoranlagen wird die Konsistenz von Datenbasen dadurch unterstützt, daß sog. kritische Abschnitte unter Interruptsperrre (Prozessorzugssperre) ablaufen. Dadurch wird erreicht, daß Programmstücke, die wichtige Datenbereiche manipulieren, zu Ende laufen können und exklusiven Zugriff zu diesen Daten haben, bis diese wieder in einem konsistenten Zustand sind.

Bei Mehrprozessorsystemen muß ein solcher Datenbereich während der kritischen Phase vor dem Zugriff der übrigen Prozessoren geschützt werden. Dafür benötigt man Sperrgrößen, die den Zugriff auf die Datenbereiche oder auch komplexere Funktionsabläufe regeln, wie sie etwa durch Pfadausdrücke beschreibbar sind /CamR74/.

Ein Prozessor kann nun gerade dann ausfallen, wenn er solche Sperrgrößen belegt hat und die Daten sich in einem inkonsistenten Zwischenzustand befinden.

Wurde die Problematik bei der Konzeption nicht berücksichtigt, so kann durch den Ausfall eines einzigen Prozessors das gesamte System blockiert werden. Für genauere Aussagen braucht man hier die Wahrscheinlichkeit, mit der sich ein Prozessor in einem solchen kritischen Abschnitt befindet. Sicher ist, daß bei Mehrprozessorsystemen ein höherer Synchronisationsaufwand nötig ist und dadurch auch das Verhältnis kritisch zu unkritisch höher sein kann.

2.2 Komplexität von Mehrprozessorbetriebssystemen

Die genannten zusätzlichen Probleme bei Mehrprozessorsystemen zeigen, daß die Gesamtkomplexität gegenüber einer Einprozessoranlage sicher nicht abnimmt.

Ausgehend von Umfang und Anlaufschwierigkeiten bestehender Einprozessorsysteme, erscheint es unmöglich, mit der an Hochschulen vorhandenen Kapazität ein neues Betriebssystem mit entsprechendem Komfort für Mehrprozessoranlagen in vertretbarer Zeit zu erstellen.

Um Entwurf und Implementierung über die üblichen Kernfunktionen hinaus zu realisieren, muß man versuchen, das Mehrprozessorbetriebssystem durch weitgehende Entflechtung und Modularisierung zu vereinfachen. Diesem Vorhaben kommt die in 1.1 zitierte Prognose entgegen, wonach Systeme der Zukunft asymmetrisch und hierarchisch sein werden. Für die einzelnen unterschiedlichen Rechner werden in größerem Umfang schon Betriebssystemkomponenten aus Monosystemen vorhanden sein.

Man kann deshalb für das in 1 beschriebene Mehrprozessorsystem folgenden Betriebssystementwurf versuchen:

Die einzelnen Rechnermoduln besitzen lokale Betriebssysteme, die wie die Rechner selbst nicht gleich sein müssen, z.B. könnten sie unterschiedlichen Funktionsumfang, unterschiedliche Benutzerschnittstellen oder auch unterschiedliche Prozeßkontrollblöcke haben.

Gemeinsam für alle Rechnermoduln müssen Funktionen existieren für die gegenseitige Synchronisation und zur Vermittlung von Aufträgen zwischen und innerhalb des einzelnen Moduls. Auf höherer Ebene wird man weitere gemeinsame Funktionen (Typen) brauchen, z.B. für Jobabwicklung und Dateisysteme. Solange solche Funktionen auf lokalen Rechnern ablaufen, müssen sie nicht notwendig gleich implementiert sein, noch müssen auf jedem Modul alle Funktionen vorhanden sein.

Ein so entflochtenes modulares System ist wesentlich weniger komplex, und die einzelnen lokalen Systeme lassen sich getrennt und unter Verwendung schon vorhandender Komponenten erzeugen. In einem System mit Arbeitsteilung lassen sich manche Typen auf bestimmte Rechner beschränken und können dadurch auch leichter optimiert oder veränderten Bedingungen angepaßt werden. Die Modularisierung löst oder mindert auch das bei Mehrprozessoren auftretende Problem, daß analog dem Engpaß "gemeinsamer Speicher" bei zentralisierten Systemen Zugriffskonflikte an gemeinsamen Datenbasen auftreten können /NehJ72/.

Die genannte Verteilung und Modularisierung muß selbstverständlich durch höheren Laufzeitaufwand bezahlt werden. Man sollte aber als wesentlichen Vorteil von Mehrprozessoranlagen hervorheben, daß gerade die größere Anzahl von oft billigeren aber für Verwaltungsfunktionen ausreichenden Prozessoren einen höheren Laufzeitaufwand zulassen, der eine Entflechtung und alle damit verbundenen Vorteile ermöglicht. Als Beispiel eines Systems mit einer Anzahl "einfacher" Rechner für Verwaltungsaufgaben können die Großrechner der Firma CDC (Cyber und Vorgänger) angeführt werden /ThoJ70/.

Wie Bild 2.1 zeigt, das die Aufrufhierarchie /ParD72a/ in Form eines Schichtenmodells darstellt, erhält man neben der üblichen vertikalen Schichtung zusätzlich eine horizontale Struktur, in der sich die unterschiedlichen Komponenten der einzelnen Moduln darstellen lassen. Da bei diesen Strukturen gemeinsame Typen nicht notwendig in Basisschichten liegen, d.h. tiefere Schichten modulspezifisch unterschiedlich sein können, kann eine Spezifikation und Implementierung nicht streng in Reihenfolge der Schichtung erfolgen. Der im weiteren beschriebene gemeinsame Typ PORT zur Auftragsvermittlung z.B. ruft Synchronisationsfunktionen auf, die sich auf irgend eine Form des Prozeßbegriffes bezieht. Dies heißt aber, daß für die einzelnen Moduln bestimmte, in der Aufrufhierarchie tiefer liegende, Funktionen schon durch den aufrufenden Typ spezifiziert sind. Man könnte dies wie in Bild 2.2 skizziert, durch eine neue Hierachierelation beschreiben, etwa als Spezifikationshierarchie bezeichnet.

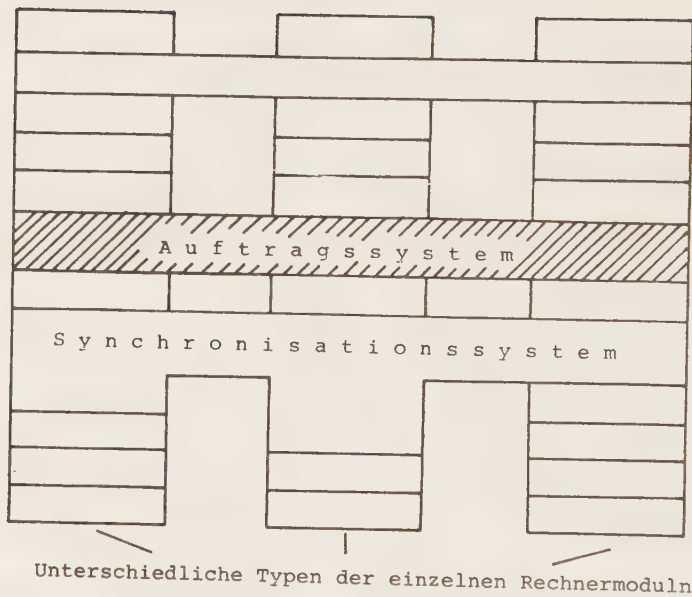


Bild 2.1: Auftragssystem innerhalb einer Aufrufhierarchie



Bild 2.2: Auftragssystem innerhalb einer Spezifikationshierarchie

3. Auftragssystem für eine Mehrprozessoranlage

3.1 Problemstellung

In einer Mehrprozessoranlage kommt dem Auftrags- und Kommunikationssystem eine zentrale Bedeutung zu.

Wie in Kap.1 und 2 ausgeführt wurde, ist es durchaus sinnvoll, daß Mehrprozessoranlagen aus unterschiedlichen Prozessoren mit teilweise unterschiedlichen Betriebssystemkomponenten bestehen können. Soll jedoch zwischen den verschiedenen Prozessoren eine sinnvolle Zusammenarbeit ermöglicht werden, so muß Einigung über die Art der Kommunikation herrschen, d.h. man braucht einen entsprechenden Datentyp, dessen Spezifikation für alle "lokalen" Systeme innerhalb der Mehrprozessoranlage identisch ist.

In der vorliegenden Arbeit wird von einem Auftragssystem und nicht vom neutraleren Kommunikationssystem gesprochen, weil in den meisten Fällen eine Kommunikation zwischen Prozessen doch dem Zwecke einer Auftragsübergabe dient, z.B. Lesen und Schreiben von internen oder externen Datenbeständen oder Ausführen von speziellen Funktionen auf mitgelieferten Datenbereichen.

3.2 Mögliche Lösungswege

Zunächst kann man Systemdienste auf zwei Arten realisieren. Man stellt Prozeduren zur Verfügung, die vom jeweiligen Auftraggeber, meist über Spezialbefehle (SVC) angesprochen, durchlaufen werden (abstrakter Datentyp, Domändenumschaltung) oder man installiert eigene Prozesse und übergibt Systemdienste als Aufträge an solche Prozesse. Nach der ersten Methode werden Systemdienste bei vielen bestehenden Einprozessorsystemen durchgeführt, da dieser Weg im allgemeinen weniger Verwaltungsaufwand erfordert (z.B. keine Prozeßumschaltung). Grundsätzlich ließe sich diese Methode auch auf Mehrprozessoranlagen übertragen, z.B. /CasL77/.

Andererseits bietet sich aber gerade bei Mehrprozessorsystemen eine Abwicklung in parallel laufenden eigenen Prozessen an, die auf einem für den entsprechenden Dienst besonders gut geeigneten Prozessor ablaufen.

Beim Aufruf von Systemdiensten müssen entsprechende Aufträge an andere Prozesse gesandt werden. Eine einfache Möglichkeit zur Auftragsübergabe wird im sog. Modellbetriebssystem /DeTrJ72/ beschrieben. Der Auftragsaustausch erfolgt dort über ein neutrales Nachrichtensystem, d.h. der Auftraggeber schickt eine Nachricht, die den Auftrag enthält, an den zuständigen Prozeß. Bei Auftragsende schickt der Auftragnehmer wieder eine als Quittung zu interpretierende Nachricht zurück.

Dieses reine Nachrichtensystem erweist sich aber als zu unflexibel, um z.B. speziellere Funktionen, wie 'Warten auf einen ganz bestimmten Auftrag' oder 'ein bestimmtes Auftragsende' zu realisieren. Bei realen Rechnern enthalten die Nachrichtensysteme deshalb meist Modifikationen wie 'Warten auf eine Nachricht von einem bestimmten Prozeß' oder 'mit einer bestimmten Kennung'.

Einen weiteren Nachteil des reinen Nachrichtensystems bildet die Deadlockgefahr, die dadurch entstehen kann, daß für die Rückantwort kein ausreichender Speicherplatz vorhanden ist.

Im folgenden wird deshalb ein Auftragssystem für Prozesse in einer Mehrprozessoranlage vorgestellt, das die Nachteile des reinen Nachrichtensystems vermeidet aber keine Einschränkungen bezüglich Auftragsklassen und Synchronisationswünsche enthält.

3.3 Forderungen an eine Lösung

Neben allgemeinen Forderungen wie Benutzerfreundlichkeit, Leistungsfähigkeit, Korrektheit usw. wurden an das Auftragssystem folgende Forderungen gestellt:

- Das System soll auf einer Mehrprozessoranlage mit unterschiedlichen Prozessoren lauffähig sein.

- Dienste (Service-Prozesse) sollen auch während der Laufzeit zu- oder abgeschaltet werden können.
- Bei Bedarf sollen für einen Dienst mehrere parallel laufende Prozessoren einsetzbar sein (auch dynamisch änderbar).
- Benutzer- wie Serviceprozesse sprechen Dienste über logische Namen an, die erst dynamisch zugeordnet werden müssen.
- Verbindungsaufbau und Auftragsvergabe sollen getrennte Funktionen sein (Line-switching).
- Das Auftragssystem soll Aufträge bis zum Auftragsende verwalten und über den jeweiligen Status Auskunft geben können.
- Um Aufwand und Deadlockprobleme zu vermeiden, soll das Auftragssystem möglichst nicht ständig Speicherplätze anfordern und freigeben.
- Es sollen alle notwendigen Synchronisationsmöglichkeiten zur Verfügung stehen (z.B. OR-Wait)
- Das Auftragssystem soll in ein Objektschutzssystem integrierbar sein, soll dieses aber für die Grundfunktion nicht voraussetzen.
- Es sollen Mechanismen eingebaut werden, die Überwachung und Test unterstützen.
- Es sollen Funktionen zur Durchsatz- und Engpaßmessung vorhanden sein.

3.4 Lösungsidee

Das vorliegende Kapitel soll einen Überblick über die gesamte Lösung vermitteln. In den folgenden Kapiteln werden dann die Datenobjekte und einzelnen Funktionen beschrieben.

Kapitel 3.7 behandelt spezielle Merkmale und gibt dadurch im Nachhinein genauere Begründungen für einzelne Entscheidungen.

3.4.1 Überblick

Auftragskanäle sind gerichtet, d.h. die Aufträge laufen von einem Prozeß - hier als User-Prozeß bezeichnet - zu einem als Service-Prozeß bezeichneten Auftragnehmerprozeß.

Die Verbindung zwischen User und Server könnte für jeden Auftrag neu aufgebaut werden oder für längere Zeit bestehen bleiben, ähnlich dem packetswitching oder lineswitching in Daten-netzen. Beim 'lineswitching' braucht man eigene Funktionen zum Verbindungsauf- oder -abbau und Speicherstrukturen, die bestehende Verbindungen beschreiben. Dabei müssen beim Verbindungsaufbau folgende Funktionen ablaufen:

- Suchen des zugehörigen Auftragskanals über symbolische Bezeichnungen
- Überprüfen, ob der Auftraggeber ausreichende Zugriffsrechte besitzt.
- Überprüfen, ob der Auftragskanal die gewünschten Funktionen realisieren kann.

Beim 'packetswitching' werden diese Funktionen bei jedem Auftrag mit ausgeführt.

Die Entscheidung, welches Verfahren vorteilhafter ist, hängt natürlich von der Struktur der Auftragsströme ab. Da in Anlehnung an bestehende Systeme angenommen wird, daß sich länger-dauernde Auftragsströme zwischen Prozessen ausbilden (z.B. E/A-Aufträge), wird das dem lineswitching entsprechende Verfahren gewählt.

D.h. man richtet mit speziellen Funktionen Auftragskanäle zwischen Prozessen ein, über die sie dann Aufträge abwickeln können. Erst wenn ein Prozeß keine Aufträge mehr senden oder empfangen will, löst er (oder auch ein anderer Prozeß) die Verbindung zu diesem Auftragskanal wieder auf.

Wie schon erwähnt, sollen über einen Auftragskanal mehrere Prozesse Aufträge senden und gleichzeitig an einen Auftragskanal im Bedarfsfall mehrere Auftragnehmerprozesse angeschlossen werden können. Dies bedeutet, Auftragskanäle gehören nicht zu einem einzelnen Prozeß und können eine längere Lebensdauer als die angeschlossenen Prozesse haben. Sie werden deshalb durch zum Auftragssystem gehörige Datenelemente beschrieben. Diese den Auftragskanal beschreibende Datenstruktur erhält den Namen PORT und das gesamte Auftragssystem wird deshalb auch als abstrakter Datentyp PORT bezeichnet.

Zum Anschluß an einen solchen PORT besitzt ein Prozeß sog. Logical Unit-Elemente, mit denen er als Auftraggeber (User) oder Auftragnehmer (Server) angeschlossen werden kann.

Im nachfolgenden Text wird zur besseren Unterscheidung eine LU (Logical Unit), wenn sie als Auftragsquelle (USER) dient, als ULU bezeichnet und als SLU, wenn sie eine Auftragssenke (Server) darstellt. Ein Prozeß kann gleichzeitig ULUs und SLUs besitzen.

Ein Prozeß spricht LUs intern über Nummern an oder, abhängig vom Programmiersystem, auch über frei wählbare Namen, die an der Schnittstelle zum PORT in Nummern übersetzt sein müssen. Die Zuordnung von LUs zu einem Auftragskanal (E/A-Gerät) erfolgt erst zur Laufzeit, vor Programmstart oder auch erst vor Aufruf des Systemdienstes, abhängig vom jeweiligen Systemdienst und der Betriebsmittelvergabe-strategie eines Systems. Die dynamische Zuordnung von 'logical unit' zu einem Auftragskanal ist auf der User-Seite bei vielen bestehenden Systemen üblich. Neu ist aber beim vorliegenden Auftragssystem, daß diese Flexibilität auch auf der Service-Seite vorhanden ist, d.h. auch Service-Prozesse werden dynamisch Auftragskanälen zugeordnet. Man erhält dadurch die folgenden zusätzlichen Möglichkeiten:

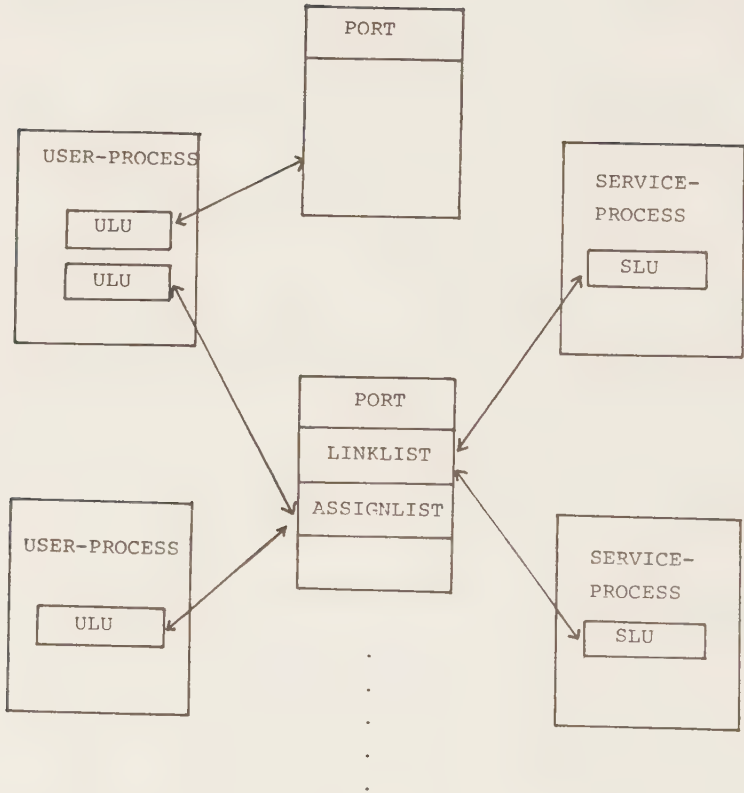


Bild 3.1: Konfigurationsbeispiel

- Neue Systemdienste können auch zur Laufzeit eingebracht werden (z.B. Anschluß neuer E/A-Geräte).
- Im Bedarfsfall kann die Anzahl der Auftragnehmerprozesse angepaßt werden (z.B. bei Überlastung könnte man ein System durch zusätzliche Prozessoren mit zusätzlichen Serviceprozessen erweitern).
- Bei Ausfall eines Prozessors kann man den Auftragsstrom auf einen anderen Prozeß (Prozessor) umleiten (ein notwendiger Mechanismus zur Erhöhung der Ausfallsicherheit).

Bild 3.1 zeigt eine mögliche Konfiguration von PORTs und LUs. In Kap. 3.4.2 wird zur weiteren Erläuterung der Ablauf eines Auftrages skizziert.

3.4.2 Ablauf eines Auftrages

Gibt ein Prozeß einen Auftrag ('Order') an das System, so werden die Auftragsdaten aus einem Parameterblock nach entsprechender Prüfung in das zugehörige ULU-Element übertragen und in die Order-list am PORT eingehängt. Analog kann ein Service-Prozeß über eine SLU freie Aufträge entgegennehmen. Ein übernommener Auftrag wird weiterhin vom Port verwaltet und erst bei der Endmeldung durch den Serviceprozeß aus der Liste gestrichen. User- und Serviceprozeß synchronisieren sich über Semaphoren in den LUs, wobei durch eine TEST-Funktion und ein im Typ Semaphore realisiertes OR-Wait die notwendige Flexibilität erreicht wird. In der ULU bzw. SLU steht nur die Beschreibung des Auftrages. Für zu transferierende oder anderweitig benötigte Daten werden Puffer-Capabilities übergeben. Das Auftragssystem sorgt dafür, daß diese Capabilities bei Auftragsende wieder zurückgegeben werden.

Hier wäre es auch denkbar, daß das Auftragssystem zusätzliche 'Utility'-Funktionen zur Verfügung stellt, mit denen der Service-Prozeß Daten zwischen eigenen Speichern und solchen übergebenen Puffern transferieren läßt.

3.4.3 Diagnose und Bewertungsfunktionen

Da man nicht völlig korrekte und funktionssichere Hard- und Softwaremechanismen voraussetzen kann, muß das Auftragssystem die einzelnen Aufträge überwachen. Zu diesem Zweck verbleiben Aufträge bis zu ihrem Ende in der Buchführung des Typs PORT. Das Auftragsende wird durch eine eigene Funktion realisiert. Zur zeitlichen Überwachung werden der jeweilige Auftragsbeginn und die maximale Auftragsbearbeitungszeit vom Typ PORT verwaltet. Die Überwachung muß von übergeordneten Überwachungsprozessen durchgeführt werden, die dann im Fehlerfall den Auftragsstrom auf andere Prozesse oder Prozessoren umschalten können.

Neben der Erkennung von Fehlverhalten ist speziell beim Auftragsystem eine Durchsatzbewertung sehr wichtig, (z.B. Engpässe oder Leerlauf von Prozessoren). Da man häufig zu wenig Vorkenntnisse über zu erwartende Auftragsströme hat, ist man für genauere Untersuchungen auf Messungen angewiesen. Man braucht deshalb Datenelemente und Mechanismen, die z.B. die Ermittlung der Warteschlangenlänge, mittlere Bedienzeiten und Auslastungen zulassen. Auf Grund dieser Werte kann z.B. die Anzahl der Serviceprozesse angepaßt werden.

3.5 Daten-Objekte des Auftragssystems

Das Auftragssystem enthält zwei unterschiedliche Datenstrukturen: PORTs, als Repräsentanten von Auftragskanälen, und die zu Prozessoren gehörenden LUs (Logical Units), über die Auftraggeber und -nehmer an einen PORT angeschlossen werden.

Ein PORT enthält folgende Informationen:

- PORT-Name
- Owner Identifikation des Erzeugerprozesses
- Attribute geben an, welche Art und welche Funktion von Aufträgen der PORT vermitteln kann.

- ACCESSLISTE Liste der zum Anschluß autorisierten Prozesse oder Prozeßklassen
- Listen der angeschlossenen Auftraggeber und Auftragnehmer
- Auftragswarteschlange
- Diagnose und Statistikinformationen, die z.B. einen Time-out erkennen oder mittlere Warteschlangen berechnen lassen.

Die PORT-Namen sind in einer Liste enthalten, wobei eine globale Liste und in jedem Rechnermodul je eine lokale PORT-Liste existiert.

Abgeleitet von einem Dateisystem werden PORTs nach Geräte- oder Volume-Port und Datei-Port unterschieden, wobei Dateiports nur zusammen mit dem zugehörigen Volume-Port existieren können.

Ein PORT-Name hat die Form:

V-Name:Dt-name. Ist der Dt-name (Dateiname) leer, so handelt es sich um einen Geräte- oder Volume-Port. Datei-Port kann hier auch als ein beliebiger 'Teildienst' eines Systems aufgefaßt werden.

Eine LU enthält folgende Information:

- LUNR Nummer (Kennung) der LU
- PORTNAME Name des PORTs an den sie angeschlossen ist
- ATTRIBUT Angaben über gewünschte/angebotene Funktionen
- ORDERSTATE Zustand des Auftrages
- Auftragsbeschreibungsdaten (Funktion und Puffercapability)
- ENDSEM Synchronisationsvariable
- Diagnosedaten, z.B. Orderbeginn zum Erkennen von Time-outs

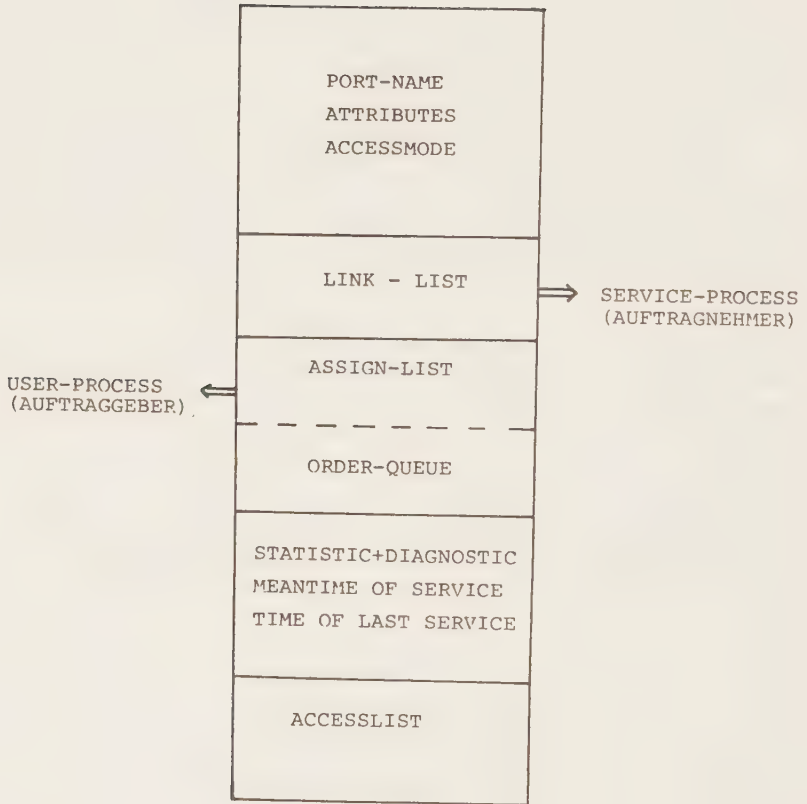


Bild 3.2: Datenelement PORT

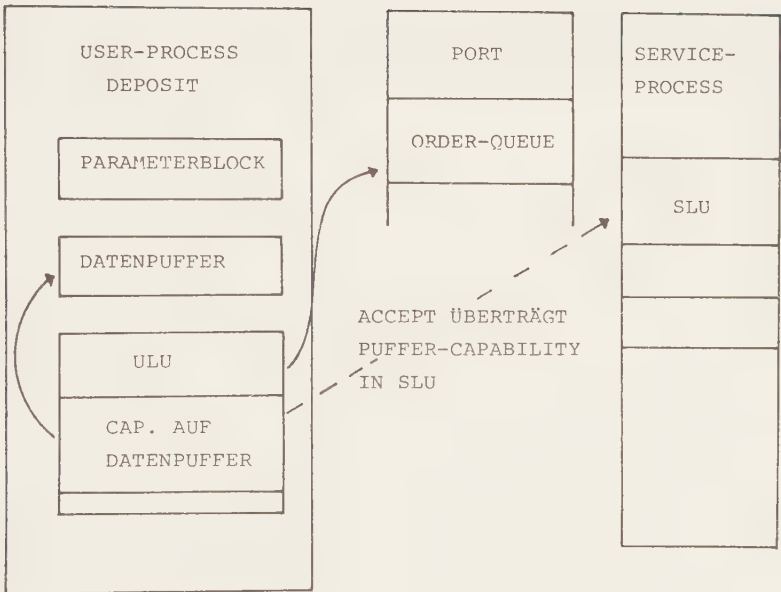


Bild 3.3: Verschiedene Datenstrukturen des Typs PORT

Ein Prozeß besitzt ein Feld von LU, wobei LUs beliebig als ULU (Auftragsquelle) oder SLU (Senke) verwendet werden können. LUs besitzen maximal die Lebensdauer des zugehörigen Prozesses.

Auf PORTs und LUs hat, bedingt durch das gewählte Schutzmodell, der einzelne Auftraggeber oder Auftragnehmerprozeß keinen direkten Zugriff. Deshalb werden bei der Auftragsvergabe oder Annahme Parameterblöcke mitgeliefert, über die im wesentlichen Funktionscode, Puffercapability und Status übergeben werden.

Bild 3.2 und 3.3 geben einen Überblick über die verwendeten Datenbereiche.

3.6 Funktionsumfang

Die Funktionen lassen sich in 5 Gruppen einteilen:

- Funktionen zum Erzeugen und Löschen von Auftragskanälen (PORT)
- Funktionen zum Auf- und Abbau von Verbindungen
- Funktionen zur Abwicklung der Aufträge
- Funktionen zur Koordinierung
- Funktionen, die der Diagnose und Bewertung dienen

Daneben ist noch eine Unterscheidung möglich, ob Funktionen auf der Auftraggeberseite (user) oder Auftragnehmerseite (server) oder gemeinsam benutzt werden. Neben von außen aufrufbaren Funktionen enthält das System noch eine Menge von nur intern bekannten Hilfsfunktionen. Bild 3.4 enthält die nach außen bekannten Funktionen, die in den folgenden Kapiteln näher beschrieben werden.

Wie später noch näher erläutert wird, existieren in der Mehrprozessoranlage in jedem einzelnen Computer-Modul lokale Auftragskanäle, und für Aufträge zwischen Modulen globale Auftragskanäle. Während Funktionen zur Bearbeitung der lokalen Kanäle vom jeweiligen Prozessor selbst durchgeführt werden, dürfen die globalen Kanäle aus den schon erwähnten Sicherheitsaspekten nur vom sog.

User-Processes

Service-Processes

Init Portlist

Create Port

Create LU-Area

Assign/Unassign

Link/Unlink

Deposit/Cancel

Accept/Reject

Reply

Delete Port

Delete LU-Area

WAITLU

TESTLU

ORLU/UNORLU

PRI-UNASSIGN

PRI-UNLINK

Bild 3.4: Funktionen des Typs PORT

PORT-Rechner bearbeitet werden. Für die beteiligten Prozesse bleibt dies natürlich im wesentlichen verborgen, nur beim Erzeugen von PORTs und beim Verbindungsaufbau muß aus dem PORT-Namen ersichtlich sein, ob es sich um einen globalen oder lokalen PORT handelt. Die folgenden Beschreibungen gelten im allgemeinen für beide Typen von PORTs.

3.6.1 Funktionen zum Erzeugen und Löschen der für Auftragskanäle (PORTs) nötigen Datenstrukturen

- CREATE - PORT
- DELETE - PORT
- CREATE - LU-AREA
- EXPAND - LU-AREA
- DELETE - LU-AREA

(GETNEW-PORTNAME, SETLU-ATTRIBUT)

CREATE (DELETE) erzeugt bzw. löscht einen Auftragskanal in Form eines Datenelementes PORT, wobei in jedem Computermodul je eine Liste von lokalen PORTs existiert und im PORT-Rechner eine Liste mit den globalen PORTs. Um spätere Aufrufe der Speicherverwaltung und um Diagnose und Fehlereingrenzung beeinträchtigende Listenverkettungen zu vermeiden, wird im PORT Platz für die Verbindungslisten (Assign-List, Link-List) und die Auftragswarteschlangen bereitgestellt. Die Assign-Liste und die Liste für Auftragswarteschlangen (Order-List) haben gleiche Länge. Da über jede Verbindung nur jeweils ein Auftrag in Bearbeitung sein kann, können bei der Auftragsabwicklung (d.h. zur Laufzeit) keine Speicherengpässe mehr auftreten.

Die festen Listenlängen sind keine Einschränkungen, da man genügend Kenntnis über zu erwartende Auftragsvolumina oder wenigstens sinnvolle Warteschlangenlängen haben sollte, oder diese über die integrierten Meßfunktionen erhalten kann.

Eine wichtige, aber in dieser Arbeit nicht weiter verfolgte Frage ist, wer die CREATE-Funktion aufrufen darf. Sicher soll ein Auftragnehmerprozeß PORTs einrichten können, aber auch z.B. Bedienprozesse oder entsprechend privilegierte Überwacherprozesse müssen PORTs erzeugen können. Jeder PORT enthält die Identifikation des Erzeugerprozesses. Es wird in jedem System eine gewisse Anzahl Standard-PORTs geben (Standard-Geräte, Standard-Dienste), die nach dem Umladen erzeugt oder mit entsprechenden Programmiersystemen schon vorher generiert werden müssen.

Die Funktion DELETE löscht PORTs wieder, wobei zu diesem Zeitpunkt natürlich keine Verbindungen mehr vorhanden sein dürfen. Die Funktion GET-NEW-PORT-Name liefert jeweils einen neuen systemeindeutigen PORT-Namen, sie dient der Lösung folgenden Problems: Sollen zwei oder mehrere Prozesse Aufträge (Nachrichten) austauschen, so kann dafür von einem übergeordneten Prozeß (z.B. Bedienprozeß) ein PORT eingerichtet und die Prozesse über diesen verbunden werden, ohne daß sie den PORT-Namen oder die Prozeßidentifikation des anderen kennen müssen. Die genannte Funktion liefert in einfacher Weise ohne Namenskonflikt dafür jeweils einen PORT-Namen.

Die Funktion CREATE(DELETE) LU-AREA erzeugt (löscht) den zum einzelnen Prozeß gehörenden Datenbereich mit LU-Elementen, über die Verbindungen aufgebaut und Aufträge abgeschickt oder entgegengenommen werden. Die LU-Area wird vom Ladeprozeß oder vom Prozeß selbst erzeugt, oder sie ist im Rahmen eines entsprechenden Programmiersystems schon im Ladeobjekt vorhanden. Daß der Prozeß selbst diesen Bereich nicht direkt manipulieren kann, wird vom zugrunde gelegten Capability- und Domänenkonzept garantiert.

Mit der Funktion EXPAND LU-AREA läßt sich im Bedarfsfall der LU-Bereich vergrößern. Dies ist sinnvoll, da z.B. bei einem, ein Dateisystem bearbeitenden Serviceprozeß die Anzahl gleichzeitig eröffneter Dateien nicht von Anfang an bekannt ist.

Mit SET-LU-ATTRIBUT lassen sich gewünschte Auftragskanaltypen und zugehörige Funktionen in eine LU eintragen. Beim Anschließen an einen PORT kann dann geprüft werden, ob dieser die notwendigen Eigenschaften besitzt, d.h. man erkennt solche Fehler nicht erst bei Auftragsvergabe.

3.6.2 Funktionen zum Auf- und Abbau von Verbindungen

- Assign/Unassign
- Link/Unlink
- PRI-Unassign
- PRI-Unlink

Ein Prozeß, der Aufträge senden oder empfangen will, braucht einen prozeßspezifischen Datenbereich, die LU-Area, ein Feld von LU-Elementen (Logical Unit). Die LUs werden über LU-Nummern i ($1 \leq i \leq \text{LU-max}$) angesprochen. Eine LU kann als Auftragsquelle (ULU=User Logical Unit) oder als Auftragssenke (SLU=Service Logical Unit) verwendet werden, entsprechend wird sie über Assign oder Link an einen PORT angeschlossen.

An einem PORT können, je nach Typ, mehrere ULUs und auch SLUs angeschlossen werden. Aufträge laufen über eine gemeinsame Warteschlange (FIFO-oder prioritätengesteuert) und werden auf der Serviceseite an freie Prozesse in Round Robin Strategie vergeben.

Die genannten Funktionen können vom Prozeß selbst und von, auf dem gleichen Prozessor laufenden, übergeordneten Prozessen (z.B. Bedienprozeß) aufgerufen werden. Dabei wird geprüft, ob ein Serviceprozeß die im PORT verzeichneten Funktionen durchführen kann, bzw. ob der PORT die vom User gewünschten Funktionen enthält. Die Frage, wieweit sich ein Prozeß überhaupt an einen PORT anschließen darf, wird in Kap. 3.7 "Realisierung des Objektschutzes" näher behandelt.

Beim Anschluß an einen Geräteport (Volumeport) muß der PORT bereits vorhanden sein. Ein solcher PORT wird normalerweise von der "Serviceseite" eingerichtet. Betrachtet man dagegen ein Dateisystem, so ist es sicher nicht sinnvoll, für alle Dateien im voraus PORTs einzurichten, hier geht die Initiative im allgemeinen vom User aus.

Bei "Assign" an einen nicht vorhandenen Dateiport schickt deshalb das Auftragssystem eine "OPEN"-Order an den am zugehörigen Volumeport angeschlossenen Serviceprozeß und veranlaßt diesen zum Einrichten eines Dateiports.

Die Funktionen Unassign und Unlink lösen Verbindungen wieder auf. Wenn bei Unassign die letzte ULU von einem Dateiport abgehängt wird, so wird in Umkehrung zum Assign eine "CLOSE"-Order an den zugehörigen Volumeport geschickt. Unassign und Unlink dürfen nur aufgerufen werden, wenn auf diesen Kanälen keine Aufträge in Bearbeitung sind.

Dagegen ist es mit den Funktionen PRI-Unassign und PRI-Unlink (PRI=privilegiert) möglich, zu jeder Zeit Verbindungen abzubauen. Dies ist nötig, wenn ein Prozeß oder Prozessor fehlerhaft läuft, oder eine Verbindung illegal zustande kam. PRI-Funktionen dürfen nur von entsprechend privilegierten Prozessen (z.B. Überwacher) aufgerufen werden. Auf diesem Weg ist es möglich, bei Ausfall eines Rechnermoduls Auftragskanäle auf andere Rechner umzuleiten.

3.6.3 Funktionen zum Senden und Empfangen von Aufträgen

DEPOSIT/CANCEL

ACCEPT/REJECT

REPLY

Über DEPOSIT gibt ein User einen Auftrag (Order) an den PORT. Die Auftragsdaten werden dabei in die ULU geschrieben und diese in eine Order-Warteschlange am PORT eingetragen. Sollte es aus irgendwelchen Gründen nötig sein, kann ein Auftrag über die Funktion CANCEL wieder zurückgenommen werden, solange er noch nicht in Bearbeitung ist. Auf der Auftragnehmerseite deklariert die Funktion ACCEPT eine SLU als empfangsbereit für eine ORDER (Synchronisationsfunktionen werden im nächsten Kapitel behandelt). Bei Eintreffen einer ORDER wird die sie beschreibende Information in der SLU abgelegt. Eine ORDER bleibt dabei weiterhin bis zum Auftragsende in der Buchführung des PORTs.

Mit REPLY gibt der Serviceprozeß das Auftragsende an. Die ORDER wird aus der PORT-Buchführung gelöscht. Diese Vorgehensweise verhindert Schwierigkeiten bei der Zuordnung von Aufträgen und Quittungen. Sie erleichtert auch die Überwachung.

Ein Orderelement enthält nur auftragsbeschreibende Daten. Müssen bei einem Auftrag Datenobjekte manipuliert werden, so wird im Orderelement eine Puffer-Capability übergeben. Das Auftragssystem ist mitverantwortlich, daß dem Auftragnehmer diese Capability bei REPLY wieder entzogen wird.

Es ist auch der Fall denkbar, daß Capabilities vom Auftragnehmer zum Aufrufer geschickt werden (z.B. um beim Lesen von geblockten Sätzen ein nochmaliges Umspeichern zu vermeiden). In einem solchen Fall wird eine Überwachung der Capability aber schwieriger.

3.6.4 Synchronisationsfunktion

WAITLU

TESTLU

ORLU/UNORLU

TESTLU liefert zu beliebiger Zeit den aktuellen Zustand einer LU und einer eventuell vorhandenen ORDER. Mit WAITLU kann auf der Userseite auf das Ende oder auf der Serviceseite auf das Eintreffen eines Auftrages gewartet werden.

Bei manchen Prozessen, z.B. Steuerungsprozessen oder dialogfähigen Prozessen, erscheint es sinnvoll, auf das Ende oder Eintreffen eines beliebigen Auftrages aus einer Menge von Aufträgen zu warten, und abhängig davon bestimmte Aktionen durchzuführen. Solche Probleme lassen sich zwar durch eine Menge von Prozessen lösen, indem jeder auf einen Auftrag wartet und das Ereignis an einen zentralen Prozeß weitergibt, doch erhöht dies nicht gerade die Benutzerfreundlichkeit.

Das Auftragssystem bietet daher die Möglichkeit, ein ODER-Warten durchzuführen.

Zu diesem Zweck werden LUs an eine, als Hilfs-LU verwendete, LU zu einer sog. OR-Liste verkettet. Mit ORLU/UNORLU können solche Listen erzeugt, erweitert, vermindert und vernichtet werden. Das eigentliche Warten erfolgt an der H-LU.

Beim Eintreffen eines Ereignisses erhält der Aufrufer die Nummer der LU, die geschaltet hat.

Ein explizites UND-Warten wurde nicht vorgesehen, da im Gegensatz zum ODER-Warten dafür noch keine, den hohen Aufwand rechtfertigende, Beispiele gefunden wurden.

Für die Lösung von Betriebsmittelvergabeproblemen der Klasse "Philosophenproblem" sollte ein Auftragssystem nur Unterstützung aber nicht Lösungen anbieten. (Lösungsmöglichkeit in Kap.6.5)

Das oben aufgeführte ODER-Warten läßt sich unter der Einschränkung, daß eine LU jeweils nur in einer OR-Liste sein kann, mit vertretbarem Aufwand lösen.

3.6.5 Funktionen, die der Diagnose und Bewertung dienen

Gerade bei stark parallelen Systemen ist eine Fehlersuche nach Hard- und Softwarefehlern sehr schwierig. Noch mehr als bei sequentiellen Systemen braucht man deshalb dafür unterstützende Funktionen. Ein Auftragssystem muß deshalb verschiedene zusätzliche Information mitführen, die ein Fehlverhalten erkennen lassen, z.B. den Zeitpunkt einer Auftragsvergabe um etwa Timeout zu erkennen, oder Redundanz in der Führung von Listen, um bei Ausfall von Speichermoduln eine Rekonfiguration zu unterstützen.

Neben Diagnose-Funktionen braucht man in einem Auftragssystem die Möglichkeit, Engpässe zu erkennen, um z.B. über mittlere Warteschlangenlänge oder Verlustfaktoren optimieren zu können (zusätzliche oder weniger Serviceprozesse, Listengrößen usw.).

Die vorliegende Arbeit soll auf die Notwendigkeit solcher Funktionen hinweisen. Die Datenstrukturen und internen Funktionen beinhalten Grundelemente und Lösungsansätze für diesen Problembereich. Genauere Aussagen können aber erst nach entsprechender Einsatzerfahrung gemacht werden.

3.7 Realisierung des Objektschutzes

Die Schutzmechanismen im Typ PORT basieren auf dem Objektschutzmodell nach /JonA77/ mit abstrakten Datentypen, Capabilities und Domänen. Ein vollständiger Objektschutz kann nur erreicht werden, wenn der einzelne Rechnermodul über einen Sicherheitskern die notwendigen Zugriffskontrollen ermöglicht. Aber auch bei Rechnermoduln ohne oder mit "fehlerhaften" Sicherheitskernen sorgt der Vermittlungsrechner dafür, daß Fehlzugriffe auf den jeweiligen Rechner beschränkt bleiben.

Das Objektschutzsystem garantiert zunächst, daß nur Funktionen des Typs PORT auf die zugehörigen Datenobjekte zugreifen können. Weiter sind aber innerhalb des Typs PORT die folgenden Schutzprobleme zu lösen.

- Wer darf PORT Funktionen aufrufen ?
- Wer darf PORTs generieren und löschen ?
- Wer darf sich an PORTs anschalten ?
- Welche Aufträge darf eine angeschaltete ULU senden ?
- Wie lange hat ein Service-Prozeß Zugriff auf die mit Aufträgen übergebenen Puffer ?

Nichtprivilegierte PORT-Funktionen darf in der vorliegenden Version jeder Prozeß aufrufen, die privilegierten Funktionen nur spezielle Überwacher- oder Bedienprozesse. Da die CREATE-Funktion nicht privilegiert ist, kann jeder Prozeß PORTs erzeugen. Die einzige Gefahr, die hierbei besteht, ist, daß ein Prozeß zu viele PORTs erzeugt und dadurch Speicherplatz verbraucht. Beim Erzeugen eines PORTs wird als Parameter eine Prozeßidentifikation mitgeliefert und in den PORT als sog. "Owner" eingetragen. Nur der Owner oder ein privilegierter Prozeß können einen PORT wieder löschen.

Ein zentrales Problem ist die Frage, wer sich an einen PORT anschließen darf, d.h. wer darf an wen Aufträge schicken und wer darf bestimmte Aufträge entgegennehmen. Auf unterer Ebene kann man als Antwort geben, diejenigen, die ein LINK- oder ASSIGN-Recht (Capability) für diesen PORT besitzen. Damit ist das Problem aber nicht gelöst, denn woher bekommt ein Prozeß diese Capability. Eine Frage, die bei allen Capability-betriebenen Systemen auftritt, zumindest bei Objekten, die länger leben als Prozesse.

Hier erscheint es sinnvoll, die Schutzinformation zum längerlebigen Objekt gehörig abzuspeichern, etwa in Form der bekannten Schutzmatrix. Als einfachste Lösung bietet sich an, die Zeilen dieser Matrix bei jedem Objekt als Zugriffsliste zu führen. Beim Erzeugen eines PORTs wird deshalb eine sog. Access-Liste angelegt, die angibt, welche Prozesse oder Prozeßklassen sich als Auftraggeber oder Auftragnehmer mit dem PORT verbinden können. Der Typ PORT überprüft selbst diese Zugriffsrechte.

Ein anderer Weg wäre es, die Funktionen Assign und Link nur von speziellen, privilegierten Prozessen durchführen zu lassen, die dann die Zugriffsmatrizen oder Listen verwalten würden.

Die Frage, welche Aufträge geschickt werden dürfen, wird wie folgt gelöst. PORTs und LUs enthalten als Attribute einen Auftragsstyp und eine Funktionsliste, z.B. in Form von Bitfeldern. Der Typ PORT braucht die Bedeutung des Auftragsstyps und der Funktionsnummern nicht zu kennen. Eine SLU wird nur dann angebunden, wenn ihr Funktionsumfang die Funktionsliste im PORT überdeckt und die Typen übereinstimmen. Entsprechend wird eine ULU nur angeschlossen, wenn die von ihr gewünschten Funktionen im PORT enthalten sind. Bei der Auftragsvergabe wird dann jeweils geprüft, ob die gewünschte Funktion vorhanden ist. Durch die Vorabprüfung kann verhindert werden, daß z.B. ein E/A-Gerät betrieben wird und erst später im Programm erkannt wird, daß notwendige Funktionen fehlen. In entsprechenden Programmiersystemen könnte man auch statisch prüfen, ob die gewünschten Funktionsparameter bei der Auftragsvergabe mit den, für die LU vorgesehenen, Attributen verträglich sind.

Bezieht sich ein Auftrag auf Datenobjekte, so wird mit dem Auftrag eine Capability für dieses Speicherobjekt mitgeliefert. Capabilities innerhalb eines Rechnermoduls müssen von einem entsprechenden lokalen Sicherheitskern überwacht werden.

Capabilities auf fremde (gemeinsame) Speicher werden vom PORT-Rechner verwaltet. Der PORT-Rechner kann so Zugriffe zwischen Rechnern und fremden Speichern kontrollieren, er kann aber nicht garantieren, daß die Information innerhalb des einzelnen Rechners auf den zuständigen Prozeß beschränkt bleibt. Da jeder Auftrag durch die Funktion REPLY beendet wird, kann der Typ PORT verlangen, daß dem Serviceprozeß die Puffercapability wieder entzogen wird. Wieweit dies realisierbar ist, hängt von den lokalen Schutzsystemen ab. Garantiert wird hier auch wieder nur der Entzug der globalen Capabilities, wobei zusätzlich überwacht werden muß, ob ein Auftrag überhaupt beendet wird.

Wenn das Schutzsystem es zuläßt, sollte eine Puffercapability kein Copy- sondern nur ein Move-Recht enthalten, so daß der Auftraggeber während der Auftragsbearbeitung selbst keine Zugriffsrechte mehr hat (d.h. die Daten nicht verändert oder zu bald zugreift).

3.8 Nachteile und Grenzen

Wie schon erwähnt, läßt sich ein System meist nur für ein begrenztes Anwendungsgebiet optimieren. Wenn hier von Nachteilen und Grenzen gesprochen wird, so sind dies im wesentlichen Nachteile an den Randgebieten des Anwendungsspektrums. Die vom Typ PORT angebotenen Funktionen für Objektschutz, Diagnose, Statistik, flexible Synchronisation usw. erfordern beträchtlichen zusätzlichen Aufwand, der bei sehr kleinen Systemen u.U. nicht mehr tragbar ist, d.h. dieses System erfordert eine gewisse Mindestzahl von Prozessen.

Nachteile treten auch dann auf, wenn sehr große Datenblöcke mit hoher Geschwindigkeit übertragen werden sollen. Das Auftragssystem überträgt Daten durch "Einzelspeicherzugriffe", d.h. der Initiator liest oder schreibt jedes Speicherelement einzeln und muß jedesmal die zugehörige Adresse mitliefern. Ein Blocktransfer ist aber nicht ausgeschlossen, wenn entsprechende Hardwareunterstützung vorhanden ist.

4. Formale Spezifikation der einzelnen Funktionen

4.1 Spezifikationsmethode

Im folgenden werden die einzelnen Funktionen spezifiziert. Mangels besserer Alternativen wurde als Beschreibungssprache eine PASCAL-ähnliche (WirN71) Notation gewählt, wobei zur Erhöhung der Lesbarkeit im Bedarfsfall Abweichungen in Kauf genommen wurden.

Die Semantik der Funktionen wird - soweit es sich um O-Funktionen (ParD72) handelt - durch ihre transformierende Wirkung auf die Datenstrukturen und bei V-Funktionen oder V-Anteilen durch ihre "Wirkung" auf die Ausgangsparameter beschrieben. Im einzelnen enthalten die Spezifikationen folgende Informationen:

- * Eine Beschreibung der Parameter mit Reihenfolge und Vorbesetzung
- * Im Abschnitt 'PRE:' die jeweils notwendigen oder durch die Funktion zu verändernden Eigenschaften (Zusicherungen) der beteiligten typeigenen Datenstrukturen und Parameter, wie sie vor Aufruf der Funktion gelten müssen.
- * In 'POST:' die durch die Funktion veränderten Merkmale der Daten.
- * In 'EXCEPT' Angaben über Reaktion bei Ausnahmesituationen (Fehlern).
- * In 'COMMENT' bei einigen Funktionen zusätzliche Erläuterungen

Der Punkt "Exceptions" wurde in der vorliegenden Arbeit noch nicht festgelegt. Man kann, wie bei vielen Systemen praktiziert, dem aufrufenden Modul eine entsprechende Fehlerkennung liefern, oder man könnte, wie es z.B. in (ParD76) beschrieben ist, vom Aufrufer Prozedurparameter mitliefern lassen und diese Prozeduren vom fehlererkennenden Modul aufrufen lassen.

Es existieren auch Systeme, die beide Möglichkeiten zulassen, auch mit der Variante, daß man durch eigene "Funktionen" vorab solche Prozeduren angeben kann, z.B. "On-Clause" in PL/1 oder Adressen für "Arithmetic Fault" usw. bei Start eines Programmes. Die verwendete Spezifikation benutzt die von den bekannten Verifikationsschemata übernommenen Ausdrücke Precondition und Postcondition, das Hauptgewicht wurde aber auf die Verständlichkeit der Darstellung gelegt.

Prozeduraufrufe in den Postconditions sind so zu interpretieren, daß zusätzlich die bei diesen Funktionen beschriebenen Transformationen gelten. Die angegebenen Prozeduren können Funktionen anderer Typen sein, aber auch sog. interne Funktionen des eigenen

Typs. Diese internen Funktionen werden dadurch Bestandteil der Spezifikation. Eine Implementierung ist aber im Gegensatz zu den Entry-(exportierten) Funktionen an diese Angaben nicht gebunden. Die Darstellung dient in erster Linie der Strukturierung der Spezifikation. An einigen Stellen sind die Postconditions zusätzlich auch als sequentieller Ablauf zu interpretieren dies ist aus dem Zusammenhang jeweils leicht erkenntlich.

Als Parameterwerte sind auch die folgenden Sonderwerte zulässig:

nil : undefiniert (durch default zu ersetzen)

all : alle zulässigen Werte (z.B. die Funktion soll für alle LUs eines Prozesses ausgeführt werden)

requ: Vom System soll dafür ein sinnvoller Wert eingesetzt werden (z.B. für eine Verbindung 'eine noch freie LU')

Diese Werte werden entsprechend der in (.....) stehenden Angaben ersetzt. Forderungen in den Preconditions beziehen sich dann auf die ersetzten Werte.

4.2 Datenstrukturen

4.2.1 Portlisten und Port

PORTLIST:

- TYPE - Typkennung
- GLOBOC - Kennung, ob globale oder lokale Liste
- GENNAME - laufender Name fuer GET-NEW-NAME
- PORTLIST - Liste von Portnamen und Verweisen auf zugehoerige Ports

PORT:

- TYPE - Typkennung
- PORTNAME - Name des Ports
- PROCESSID - Name des Owner Prozesses
- ATTRIBUTE - Auftragstyp und moegliche Funktionen
- MAXTIME - Maximale Bedienzeit
- ASSACCMODE - Belegmoeglichkeit und Status fuer Assign
- LINKACCMODE - Belegmoeglichkeit und Status fuer Link
- LIFETIME - Lebensdauer des Ports
- DIAGNOSTICS - Daten fuer Diagnose
- STATISTICS - Daten fuer Statistik (Bewertung)
- FILEPORT-COUNT - Anzahl der Dateiports zu diesem Port
- ASSIGNLIST - Liste der angeschlossenen ULUs
- LINKLIST - Liste der angeschlossenen SLUs
- ORDERLIST - Auftragsliste und Warteschlange (Laenge = Laenge der ASSIGNLIST)
- ACCESSLIST - Liste der zugriffsberechtigten Prozesse und Prozessklassen

4.2.2 LU-Area mit ULUs und SLUs

LUAREABLOCK:

TYPE	- Typkennung
PROCESSID	- Name des zugehoerigen Prozesses
ANZLU	- Anzahl der LUs in diesem Block
NEXTBLPTR	- Zeiger auf Erweiterungsblock
LUAREA	- Feld von LUs

ULU:

TYPE	- Typkennung 'ULU'
LUNR	- Nummer der LU
PORTNAME	- Portname, mit dem die ULU verbunden ist
ATTRIBUTE	- gewuenschte Attribute
ACCMODE	- gewuenschter Belegmodus
TIMEOUT	- spaeteste Endezeit
ORDERSTATE	- Bearbeitungszustand des Auftrages
FCCODE	- Funktionscode
STATUS	- Rueckgabe-Status
BUFCAP	- Capability auf Datenpuffer
PARPOINT	- Verweis auf Parameterblock der ULU
LUSEM	- Semaphore, zum Warten auf das Auftragsende

SLU:

TYPE	- Typkennung 'SLU'
LUNR	- Nummer der LU
PORTNAME	- Portname, mit dem die SLU verbunden ist
ATTRIBUTE	- vom Serviceprozess angebotene Attribute
ACCMODE	- Belegwunsch
MAXJOB	- Maximale Bearbeitungszeit
ORDERSTATE	- Bearbeitungszustand des Auftrages
FCCODE	- Funktionscode
STATUS	- Rueckgabe Status
BUFCAP	- Capability auf uebergebenen Datenpuffer
PARPOINT	- Verweis auf Parameterblock der SLU
LUSEM	- Semaphore zum Warten auf Auftragseingang

4.2.3 Parameterblock

PARBLK:

LUNR	- Nummer der LU (ULU/SLU)
PORTNAME	- Portname
FCCODE	- Funktionscode
STATUS	- Rueckgabe-Status
BUFCAP	- Capability fuer Datenpuffer

4.3 Exportierte Funktionen

4.3 1 INITPORTLIST (PROCID,LENGTH)

PARAMETER: PROCID - Identifikator des zugehörigen Prozessors
(if 'global then Anlegen der globalen
Portliste)
LENGTH - Länge der Portliste

PRE: 1. Portliste noch nicht angelegt,
2. Speicherplatz vorhanden

POST. PORTLIST angelegt
PORTLIST.GENNAME='|GN0001
PORTLIST.PORTLIST = 'leer

EXCEPT: if not 1 then ERROR 6
if not 2 then ERROR 7

COMMENT: In einem entsprechenden Programmiersystem kann die
Portliste und eventuell auch eine Anzahl von Standard-
PORTs schon zur Systemgenerierzeit angelegt werden.

4.3 2 CREATEPORT (PROCESSID,PORTNAME,GLOBLOC,ATTRIBUTE,ASSACC, LINKACC,LASSIGN,LLINK,LACC,IDLIST ERRVEK)

PARAMETER: PROCESSID - Identifikator des Owner Prozesses
 (if 'nil' then aufrufender Prozeß)
 PORTNAME - Name des Ports in der Form
 Vname:Dtname
 (if 'requ' then vom System erzeugter
 Name)
 GLOBLOC - Globaler oder lokaler PORT
 ATTRIBUTE - Attribute des Ports
 ASSACC - Belegmöglichkeit für ASSIGN
 LINKACC - Belegmöglichkeit für LINK
 LASSIGN - Länge der ASSIGNLIST
 LLINK - Länge der LINKLIST
 LACC - Länge der ACCESSLIST
 IDLIST - Liste von Prozeßidentifikatoren
 ERRVEK - Fehlervektor

PRE: 1. if Dtname <> nil
 then Port mit Vname: vorhanden;
 end;
 2. Port mit Vname Dtname noch nicht vorhanden;
 3. Speicherplatz für Port vorhanden;

POST: Port eingerichtet
 PORT.(PROCESSID,PORTNAME,ATTRIBUTE,
 ASSACC,LINKACC) =PARAMETER.* 1)
 PORT.(ASSIGNLIST,LINKLIST):= leer
 IDLIST elof PORT.ACCESSLIST
 PORT.rest initialisiert
 if Dateipoint then Volumeport FILEPORTCOUNT +=1;

EXCEPT: if not 1. then ERROR1;
 if not 2. then ERROR6;
 if not 3. then ERROR7;

COMMENT: Zu PRE1:Dateipoints können nur zusammen mit dem
 zugehörigen Volumeport existieren (Fileport+1).
 (X +=1 soll bedeuten X:=X+1)

- 1) Die Darstellung "A.(X,Y,Z) := B.*"
 soll eine Kurzschreibweise sein für:
 A.X := B.X
 A.Y := B.Y
 A.Z := B.Z

4.3.3 DELETEPORT (PROCESSID,PORTNAME,ERRVEK);

PARAMETER: PROCESSID - Identifikator des Prozesses
 (if 'nil' then aufrufender Prozeß)
 PORTNAME - Name des Ports in der Form
 Vname:Dtname
 (if 'all' then alle Ports die als
 Owner die angegebene PROCESSID
 enthalten)
 ERRVEK - Fehlervektor

PRE: 1. Port mit Namen PORTNAME vorhanden;
 2. keine LUs mit dem PORT verbunden

POST: Port gelöscht
 if Dateiport then Volumport.FILEPORTCOUNT -=1;
 FREESTORAGE (Port);

EXCEPT: if not 1 then ERROR1;
 if not 2. then ERROR3;

COMMENT: Durch die Möglichkeit, bei PORTNAME den Wert
 'all' zu verwenden, braucht man sich bei kurzle-
 bigen und nicht allgemeinbekannten PORTS nicht
 deren Namen zu merken.

4.3.4 GETNEWPORTNAME (GLOBAL,PORTNAME)

PARAMETER: GLOBAL - Globaler Port?
 PORTNAME - Name des Ports

PRE:

POST: PORTNAME:= G/L-PORTLIST.GENNAME
 PORTLIST.GENNAME:=SUCCESSOR(PORTLIST.GENNAME)

EXCEPT:

COMMENT: Funktion zum Erzeugen systemeindeutiger
 Port-Namen

Durch die Funktion "SUCCESSOR" soll aus
PORTLIST.GENNAME jeweils ein neuer Portname
erzeugt werden.

z.B. \$GN0002 = SUCCESSOR(\$GN0001)

4.3.5 CREATELUAREA (PROCESSID,ANZLU,ERRVEK)

PARAMETER: PROCESSID - Identifikator des Prozesses
 (if 'nil' then aufrufender Prozeß)
 ANZLU - Anzahl LUs im LU-Block
 ERRVEK - Fehlervektor

PRE: 1. LUAREA noch nicht vorhanden;
 2. Speicherplatz für LUAREA vorhanden;

POST: LUAREA mit LU-Block der Länge ANZLU existiert;
 PROCESSID in LUAREA eingetragen;
 LUs initialisiert

EXCEPT: if not 1. then ERROR2;
 if not 2. then ERROR7;

4.3.6 EXPANDLUAREA (ANZLU,ERRVEK)

PARAMETER: ANZLU - Anzahl LUs im neuen LU-Block
ERRVEK - Fehlervektor

PRE: 1. LUAREA vorhanden;
2. Speicherplatz vorhanden

POST: Zusätzlicher LU-Block der Länge ANZLU angelegt
alle neuen LUs initialisiert

EXCEPT: if not 1. then ERROR2;
if not 2. then ERROR7;

4.3.7 SETLUATTRIBUTE(PROCESSID,LUNR,ATTRIBUT,ACCMODE, PARPOINT)

PARAMETER: PROCESSID - Identifikator des Prozesses
 (if 'nil' then aufrufender Prozeß)
 LUNR - Nummer der LU
 ATTRIBUTE - Gewünschte Attribute
 ACCMODE - Belegwunsch
 PARPOINT - Pointer auf Parameterblock

PRE: 1 LU not (elof ASSIGNLIST or elof LINKLIST)

POST: LU.(ATTRIBUTE,ACCMODE,PARPOINT):= PARAMETER.*

EXCEPT: if not 1 then ERROR3;

COMMENT: Mit dieser Funktion können vor Verbindungsauf-
 bau Attribute und Belegwünsche in eine LU einge-
 tragen werden. Zusätzlich kann schon für den
 späteren Gebrauch ein Parameterblock zugeordnet
 werden.

4.3 8 DELETUAREA (PROCESSID, ANZBL, ERRVEK)

PARAMETER: PROCESSID - Prozeßidentifikator
 (if 'nil' then aufrufender Prozeß)
 ANZBL - Anzahl der zu löschenden LU-Blöcke
 (if 'all' then alle)
 ERRVEK - Fehlervektor

PRE: 1. LUAREA vorhanden
 2. alle zu löschenden LUs: not elof ASSIGN/LINK-List

POST: ANZBL LU-Blöcke (nach LIFO) gelöscht

EXCEPT: if not 1 then ERROR2;
 if not 2 then ERROR3;

4.3 9 ASSIGN (PROCESSID,LUNR,PORTNAME,ACCESS,ULUATTR,ERRVEK)

PARAMETER: PROCESSID - Identifikator des Benutzerprozesses
 (if 'nil' then aufrufender Prozeß)
 LUNR - Nummer der LU
 (if 'requ' then freie LU suchen)
 PORTNAME - Name des Ports in der Form
 Vname:Dtname
 ACCESS - Belegwunsch
 (if 'nil' then ACCESS:=ULU.ACCMODE)
 ULUATTR - Gewünschte Attribute
 (if 'nil' then ULUATTR:=ULU.ATTRIBUTE)
 ERRVEK - Fehlervektor

PRE: 1 LU not (elof ASSIGNLIST or elof LINKLIST)
 2. ULUATTR <> nil
 3 Volum-Port vorhanden
 4. PORT.LINKLIST <> leer
 5. ULUATTR in PORT.ATTRIBUTE enthalten
 (Muster wird überdeckt)
 6. Belegwunsch erfüllbar
 7. Freier Listenplatz in PORT.ASSIGNLIST vorhanden

POST: LU.TYPE:= 'ULU';
 if PORTNAME = Dateiport
 then Dateiport angelegt
 ULU elof PORT.ASSIGNLIST

EXCEPT: if not 1. then ERROR3;
 if not 2. then ERROR8;
 if not 3 then ERROR1;
 if not 4. then ERROR6;
 if not 5. then ERROR5;
 if not 6. then ERROR9;
 if not 7. then ERROR7;

COMMENT: Bei Assign an einen nicht vorhandenen Dateiport wird
 dieser angelegt.
 Zu PRE2: Nach Ausführung der Vorbesetzung.

4.3.10 UNASSIGN (PROCESSID,LUNR,PORTNAME,ERRVEK)

PARAMETER: PROCESSID - Identifikator des Benutzerprozesses
 (if 'nil' then aufrufender Prozeß)

LUNR - Nummer der LU
 (if 'all' then alle ULUs des Prozesses)
 (if 'requ' then Zuordnung über PORTNAME
 suchen)

PORTNAME - Portname
 (kann für LUNR <> requ' auch nil sein)

ERRVEK - Fehlervektor

PRE: 1. ULU elof ASSIGNLIST;
 2. ULU not elof ORDERLIST;
 3. LUNR und PORTNAME 'verträglich'

POST: ULU not elof ASSIGNLIST;
 ULU befindet sich im Initialzustand;
 if ULU last elof ASSIGNLIST eines Dateiports
 then Dateiport aufgelöscht;
 end;

EXCEPT: if not 1. then ERROR3;
 if not 2. then ERROR4;
 if not 3. then ERROR8;

4.3 11 LINK (PROCESSID,LUNR,PORTNAME,ACCESS,SLUATTR,ERRVEK)

PARAMETER: PROCESSID - Identifikator des Serviceprozesses
 (if 'nil' then aufrufender Prozeß)
 LUNR - Nummer der LU
 (if 'requ' then freie LU suchen)
 PORTNAME - Name des Ports
 ACCESS - Belegwunsch
 (if 'nil' then ACCESS:= SLU.ACCMODE)
 SLUATTR - Attribute der SLU
 (if 'nil' then SLUATTR:=SLU.ATTRIBUTE)
 ERRVEK - Fehlervektor

PRE: 1. LU not (elof ASSIGNLIST or elof LINKLIST)
 2. SLUATTR <> nil;
 3. Port mit Namen PORTNAME eingerichtet;
 4. SLUATTR mit PORT verträglich
 5. Belegwunsch erfüllbar

POST: SLU.ATTRIBUTE enthält Attribute der SLU;
 LU.TYPE:= 'SLU'
 SLU elof PORTNAME.LINKLIST

EXCEPT: if not 1. then ERROR3;
 if not 2. then ERROR8;
 if not 3. then ERROR1;
 if not 4. then ERROR5;
 if not 5. then ERROR9;

4.3 12 UNLINK (PROCESSID,LUNR,PORTNAME,ERRVEK)

PARAMETER: PROCESSID - Identifikator des Serviceprozesses
 (if 'nil' then aufrufender Prozeß)
 LUNR - Nummer der LU
 (if 'all' then alle SLUs)
 (if 'requ' then ueber PORTNAME
 ermitteln)
 PORTNAME - Name des Ports
 (kann bei LUNR <> requ' auch 'nil'
 sein)

PRE: 1. LUNR und PORTNAME verträglich
 2. Port mit Namen PORTNAME vorhanden;
 3. SLU elof LINKLIST am angegebenen Port;
 4. if SLU last elof LINKLIST
 then FILEPORTCOUNT=0 and ORDERLIST leer
 end;

POST: LU.TYPE:= 'LU'
 SLU not elof LINKLIST
 SLU ist wieder im initialisierten Zustand

EXCEPT: if not 1. then ERROR8;
 if not 2. then ERROR1;
 if not 3 then ERROR3;
 if not 4. then ERROR4;

COMMENT: Unlink soll auch dann durchgefuehrt werden, wenn der
 Prozeß schon durch "ACCEPT" seine Bereitschaft zur
 Auftragsuebernahme gemeldet hat, aber noch keinen
 Auftrag erhalten hat.

4.3.13 PRIUNASSIGN (PROCESSID,LUNR,PORTNAME,ERRORSTATE,ERRVEK)

```

PARAMETER: PROCESSID - Identifikator des Prozesses
            LUNR      - LU-Nummer
                      (if 'all' then alle ULUs)
                      (if 'requ' then über PORTNAME
                        ermitteln)
            PORTNAME  - Name des Ports
                      (bei LUNR <> 'requ' auch 'nil')
            ERRORSTATE- Fehlerstatus
            ERRVEK    - Fehlervektor

PRE:      1. Port in PORTLIST vorhanden;

POST:     if 'Auftrag elof ORDERLIST'
          then ULU.STATUS:='canceled';
            ULU not elof ORDERLIST;
            if 'Auftrag nicht in Bearbeitung'
            then ORDER not elof ORDERLIST;
            else PORT.LINKELEMENT.ORDERSTATE:='Auftrag
              canceled';
            end;
            SIGNAL (ULU.LUSEM);
          end;
          ULU not elof ASSIGNLIST
          und initialisiert;

EXCEPT: if not 1. then ERROR1;

```

4.3 14 PRIUNLINK (PROCESSID,LUNR,PORTNAME,ERRORSTATE,ERRVEK)

PARAMETER: PROCESSID - Identifikator des Prozesses
 LUNR - LU-Nummer
 (if 'all' then alle SLUs)
 (if 'requ' then über PORTNAME
 ermitteln)
 PORTNAME - Name des Ports
 ERRORSTATE - Fehlerstatus
 ERRVEK - Fehlervektor

PRE: 1. Port in PORTLIST vorhanden;

.OST: if 'Auftrag in Bearbeitung'
 then ULU.STATUS:= 'aborted';
 CAPABILITY.MOVE (SLU.BUFCAP,ULU.BUFCAP);
 SIGNAL(ULU.LUSEM);
 Vom Serviceprozeß bearbeiteter Auftrag
 in ORDERLIST gelöscht;
 end;
 SLU not elof LINKLIST
 und initialisiert;

EXCEPT: if not 1 then ERROR1;

4.3 15 DEPOSIT (LUNR,PARPOINT,PROCEED,ERRVEK)

```

PARAMETER: LUNR      - LU-Nummer
            PARPOINT - Zeiger auf Parameterblock
                      (if 'requ' then PARPOINT:=
                        ULU.PARPOINT)
            PROCEED  - Nach Auftragsvergabe nicht auf
                      Ende warten
            ERRVEK   - Fehlervektor

PRE:      1. ULU elof ASSIGNLIST;
          2. ULU not elof ORDERLIST;
          3. FCCODE im Parameterblock in ULU.ATTRIBUTE enthalten;
          4. PARPOINT <> nil

POST:     ULU.FCCODE:=PARBLK.FCCODE;
          CAPABILITY.MOVE (PARBLK.BUFCAP,ULU.BUFCAP);
          ULU.PARPOINT:=PARPOINT;
          ULU elof ORDERLIST;
          ULU.ORDERSTATE:= 'Auftrag in Bearbeitung';
          Auftragszeitpunkt in ULU eingetragen;
          if SLU in LINKLIST 'wartend'
          then MOVEORDER;
          if not PROCEED
          then WAITLU (ULU.LUSEM);
          end;

EXCEPT: if not 1 then ERROR3;
          if not 2 then ERROR4;
          if not 3 then ERROR5;
          if not 4 then ERROR8;

COMMENT:  Die Puffercapability im Parameterblock kann auch den
          Wert 'requ' haben. In diesem Fall liefert der Service
          prozess ein Capability zurueck.

```

Die Funktion "MOVEORDER" soll die Auftragsdaten von der ULU in die SLU übertragen und ein SIGNAL auf den Semaphore in der SLU ausführen

4.3.16 CANCEL (LUNR,STATUS,ERRVEK)

PARAMETER: LUNR - Nummer der LU
 (if 'all' then alle ULUs)
 STATUS - Rückgabestatus
 ERRVEK - Fehlervektor

PRE: 1. ULU elof ASSIGNLIST;

POST: if 'Auftrag in Bearbeitung'
 then STATUS:=ULU.ORDERSTATE
 (if 'Auftrag auf Bearbeitung wartend'
 then ULU not elof ORDERLIST;
 ULU.ORDERSTATE:='Auftrag canceled';
 CAPABILITY.MOVE (ULU.BUFCAP,PARBLK.BUFCAP);
 STATUS:=LU.ORDERSTATE
 end;

EXCEPT: if not 1. then ERROR3;
 if Auftrag schon ausgeführt then ERROR6;

COMMENT: Wenn der Auftragsstyp und der verwendete Typ
 'Capability' es zulassen, wird hier auch ein
 laufender Auftrag abgebrochen. (Die Schwierig-
 keit besteht darin, dem Serviceprozeß die
 Puffercapability zu entziehen)

4.3 17 ACCEPT (LUNR,PARPOINT,PROCEED,ERRVEK)

PARAMETER: LUNR - Nummer der LU
 PARPOINT - Zeiger auf Parameterblock des
 Serviceprozesses
 (if 'requ' then PARPOINT:=
 SLU.PARPOINT)
 PROCEED - WAIT auf LUSEM unerwünscht?
 ERRVEK - Fehlervektor

PRE: 1. SLU elof LINKLIST;
 2. SLU.ORDERSTATE = nil;

POST: SLU.PARPOINT:= PARPOINT
 PARBLK.LU:=LUNR;
 In LINKLIST READYSTAte für SLU gesetzt;
 SLU.ORDERSTATE:='Annahmefähig';
 In SLU Zeitpunkt der Bereitschaft zur
 Auftragsübernahme eingetragen;
 if ORDER in ORDERLIST wartend
 then MOVEORDER;
 end;
 if not PROCEED
 then WAITLU (SLU.LUSEM);
 end;

EXCEPT: if not 1 then ERROR3;
 if not 2. then ERROR4;

COMMENT: Durch ACCEPT teilt der Serviceprozeß zunächst
 nur die Bereitschaft zur Auftragsübernahme mit
 Erst wenn ein Auftrag eintrifft werden entsprechende
 Daten in die SLU geschrieben.

4.3.18 REJECT (LUNR,ERRVEK)

PARAMETER: LUNR - Nummer der LU
 (if 'all' then alle SLUs)
 ERRVEK - Fehlervektor

PRE: 1. SLU.ORDERSTATE:= 'Auftrag in Bearbeitung';

POST: ORDER an den Beginn der Warteschlange
 eingehängt;
 SLU.ORDERSTATE:= 'Auftrag rejected';
 CAPABILITY.MOVE(SLU.BUFCAP,ULU.BUFCAP);
 SLU initialisiert

EXCEPT: if not 1. then ERROR4;

4.3 19 REPLY (LUNR,ERRVEK)

PARAMETER: LUNR - Nummer der LU
 ERRVEK - Fehlervektor

PRE: 1. SLU.ORDERSTATE:= 'Auftrag in Bearbeitung';

POST: if ORDER elof ORDERLIST
 then ULU.STATUS:=SLU.STATUS;
 CAPABILITY.MOVE(SLU.BUFCAP,ULU.BUFCAP);
 ULU.ORDERSTATE:= 'Auftrag beendet';
 SIGNAL (ULU.LUSEM);
 end;
 ORDER not elof ORDERLIST;
 SLU.ORDERSTATE:= 'Auftrag beendet';

EXCEPT: if not 1. then ERROR4;

COMMENT: REPLY muß pruefen, ob der Auftrag nicht
 'canceled' wurde, deswegen die Abfrage
 'if ORDER elof ORDERLIST'

4.3.20 TESTLU (PROCESSID,LUNR,STATUS,ERRVEK)

PARAMETER: PROCESSID - Identifikator des Prozesses
 (if 'nil' then aufrufender Prozeß)
 LUNR - Nummer der LU
 STATUS - Auftragsstatus
 ERRVEK - Fehlervektor

PRE:

POST: STATUS:=LU.ORDERSTATE;

EXCEPT:

4.3 21 ORLU (H-LU, LUNR, ... LUNRn, ERRVEK)

PARAMETER: H-LU - Hilfs-LU
 LUNR1 - Nummer der LU1
 .
 .
 LUNRn - Nummer der LUn
 ERRVEK - Fehlervektor

PRE: 1 H-LU and LUi not elof ORDERLIST;

POST: LUSEM der LUi sind am Semaphore LUSEM
 der Hilfs-LU eingehängt;

EXCEPT: if not 1. then ERROR4;

COMMENT: Mit ORLU können auch einzelne LUs in eine schon
 vorhandene OR-Liste eingehaengt werden.

Die Hilfs-LU erhält als Typkennung den Wert 'HLU'.
 Es ist hier nicht festgelegt ob "OR-WAIT" vom Typ
 SEMAPHORE oder vom Typ PORT realisiert wird.
 Trifft man die Einschränkung, daß jede LU nur in
 jeweils einer OR-Liste enthalten sein darf so kann
 man diese Liste durch eine einzige Verkettung und
 einen Verweis in der LU auf die H-LU realisieren.

4.3.22 UNORLU (H-LU,LUNR1,...,LUNRn)

PARAMETER: H-LU - Hilfs-LU
 LUNR1 - Nummer der LU1
 .
 LUNRn - Nummer der LUn
 (if 'all' then OR-Kette ganz auflösen)

PRE: 1 LUi not elof ORDERLIST
 2.H-LU and LUi elof ORLIST

POST: LUSEM der LUi sind am Semaphore LUSEM
 der Hilfs-LU ausgehängt;

EXCEPT: if not 1. then ERROR4;
 if not 2. then ERROR3;

COMMENT: Mit UNORLU können auch nur einzelne LUs aus der
 OR-Kette ausgehaengt werden.

4.3.23 WAITLU (LUNR,ORDERLU,PARPOINT,ERRVEK)

PARAMETER: LUNR - Nummer der LU
 ORDERLU - Nummer der auslösenden LU
 PARPOINT - Pointer auf Parameterblock
 (if 'requ' then PARPOINT:=
 ORDERLU.PARPOINT)
 ERRVEK - Fehlervektor

PRE: 1. LU.ORDERSTATE='Auftrag elof ORDERLIST'
 oder LU ist Hilfs-LU;

POST: WAIT(LUNR.LUSEM)
 ORDERLU:=Nummer einer LU, die geschaltet hat;
 PARPOINT.(FCCODE,PORTNAME,STATUS,BUFCAP):=
 ORDERLU.*
 ORDERLU.ORDERSTATE:='Auftrag abgeholt/abge-
 geschlossen';

EXCEPT: if not 1. then WARNING

COMMENT: Da WAITLU nicht zwischen Einfach- und Order-
 warten unterscheidet muß dem Aufrufer über
 ORDERLU die LU-Nummer der geschalteten LU mit
 geteilt werden. Da er keinen Zugriff auf das LU-
 Element hat, kann er sich auch den Parameterzeiger
 liefern lassen.

Wenn PRE1 nicht erfüllt ist (z.B. kein Auftrag
 in LU) so kann dies vom Aufrufer durchaus sinnvoll
 sein, deshalb nur WARNING.

4.3.24 CHANGEACCESS (PORTNAME, RIGHTS, PROCESSID, ERRVEK)

PARAMETER: PORTNAME - Name des Ports
PROCESSID - Identifikator des Prozesses
RIGHTS - Zugriffsrechte
ERRVEK - Fehlervektor

PRE: 1. Port vorhanden;

POST: In PORT.ACCESSLIST für PROCESSID Zugriffs-
rechte geändert;
(neu eingetragen oder bei RIGHTS = nil gelöscht)

EXCEPT: if not 1. then ERROR1;

4.4 Fehlerfälle

Es wird davon ausgegangen, daß zur Programmerstellungszeit, soweit möglich, Typprüfungen durchgeführt werden. Der Fehler 'LU-Nummern zu groß' kann bei fast allen Funktionen auftreten und ist deshalb nicht jedesmal aufgeführt.

- ERROR1: Port ist nicht vorhanden oder:
 bei DELETE - Port nicht im eingerichteten Zustand
- ERROR2: LUAREA nicht vorhanden oder:
 LU-Nummer zu groß
 bei CREATELUAREA - LUAREA bereits vorhanden
- ERROR3: LU nicht mit PORT verbunden oder:
 bei ASSIGN/LINK - LU bereits verbunden
 UNLINK - noch LUs angeschlossen
 DELETE - noch LUs angeschlossen
 SETLUATTRIBUT - LU mit PORT verbunden
- ERROR4: Kein Auftrag in LU oder:
 bei DEPOSIT/UNASSIGN - bereits Auftrag in LU
 ACCEPT/ORLU - bereits Auftrag in LU
 UNLINK - noch Aufträge am PORT wartend
- ERROR5: Attribute oder Funktionscode nicht zulässig
 (z.B. Attribute in PORT überdecken nicht die
 Attribute in der ULU)
- ERROR6: Funktion zu diesem Zeitpunkt nicht zulässig oder:
 UNLINK - Im Hauptport FILEPORTCOUNT <> 0
 CREATE - Port bereits vorhanden
 ASSIGN - Port nicht im angelegten Zustand (gelinkt)
 INITPORTLIST - Portliste schon initialisiert
- ERROR7: Kein Listenplatz oder Speicherplatz vorhanden
- ERROR8: Unverträgliche Parameter
 ASSIGN/LINK - LUATTR = LU.ATTRIBUT = nil
 LU.PORTNAME <> PARAMETER.PORTNAME
- ERROR9: Belegmodus nicht möglich
 z.B. schon exklusiv belegt
- ERROR10: Schutzrechte verletzt
- ERROR11: Inkonsistenter Zustand der Daten
- ERROR12: Unzulässige Reihenfolge der Funktionen

5. Implementierungsvorschlag und Detailspezifikation

5.1 Vorbemerkungen

Die formale Spezifikation wie sie in Kap.4 ausgeführt ist, enthält alle notwendigen Informationen für einen Anwender des abstrakten Datentyps. Gemäß dem in neueren Arbeiten häufig zitierten 'hiding principle' enthält sie keine Angaben über die Abbildung der Daten auf Speicherstrukturen und den Ablauf der einzelnen Funktionen. Eine Implementierung ist an die Formalspezifikation zwingend gebunden in den übrigen Entscheidungen aber frei

Bei Mehrprozessoranlagen und speziell bei dem hier beschriebenen Auftragssystem muß aber noch folgendes Problem beachtet werden. Wie früher ausgeführt, sollen Systeme durch neue Prozessoren jederzeit erweiterbar oder änderbar sein. Auf diesen neuen Prozessoren braucht man auch Funktionen des Typs PORT. Für die Implementierung der PORT-Funktionen reicht dann aber die Formalspezifikation nicht mehr aus. Die Implementation ist hier an die schon bestehenden Datenstrukturen und Synchronisationsabläufe gebunden.

Kap.5 bringt deshalb eine detaillierte Spezifikation (Detailspezifikation) mit Beschreibungen der Datenstrukturen, Synchronisationsmechanismen und Grobabläufen aller 'nicht trivialer Funktionen.

Die Beschreibung erfolgte auch hier maschinenunabhängig in einer freien Variante von PASCAL. Es sollte nicht verschwiegen werden, daß diese Detailspezifikation auch für den reinen Anwender des Datentyps nützlich sein kann. Nach dem heutigen Stand der Spezifikationsmethoden ist es sicher nicht immer leicht alle Punkte einer Formalspezifikation zu verstehen oder alle angebotenen Möglichkeiten zu erkennen.

Ein totales 'hiding principle' scheint der menschlichen Natur und den Gepflogenheiten vieler nicht DV-Systeme zu widersprechen. (Man denke nur an einen Autofahrer, der nur nach dem 'hiding principle' zulässige Information über sein Auto hat.)

Wichtig ist sicher, daß ein Anwender nur solche Eigenschaften verwendet, die aus der Formalspezifikation ableitbar sind. Eine wichtige (leider nicht immer vorhandene) Voraussetzung dafür ist natürlich, daß die Trennung Formal-Detailspezifikation existiert.

5.2 Synchronisationsprobleme

Innerhalb des Typs PORT existieren verschiedene sog. kritische Abschnitte, d.h. Teilroutinen, in denen bearbeitete Datenelemente in einem Zustand sind, in dem sie von anderen Routinen nicht korrekt bearbeitbar sind. Z.B. darf eine LU nicht schon während des Verbindungsaufbaues mit Aufträgen belegt werden. In der Auftragswarteschlange können nicht ohne Schwierigkeiten mehrere Serviceprozesse gleichzeitig nach Arbeit suchen usw.

Da man mehrere Exemplare (Inkarnationen) der einzelnen Datenstrukturen hat und der gegenseitige Ausschluß sich auf nur ein oder einige Exemplare beschränkt, ist es sinnvoll, die notwendigen Synchronisationsinformationen bei den Datenexemplaren selbst zu führen. Jedes Datenelement enthält deshalb eine Lock-Variable (z.B. Port-Lock, LU-Lock).

Ein zusätzliches Problem tritt dadurch auf, daß Datenelemente (einzelne Ports) erzeugt und gelöscht werden können. Beim Löschen verschwindet auch die zugehörige Lockvariable. Es muß also sichergestellt sein, daß zu diesem Zeitpunkt kein Prozeß an der Lockvariablen 'wartet'.

Zur Lösung dieses Problems enthält jeder Eintrag in der Portliste (=Liste der Portnamen mit Zeiger auf das Portelement) zusätzlich einen sog. Referenzcount (REFCOUNT), der entsprechend hochgezählt oder erniedrigt wird. Seine Manipulation wird durch eine zentrale Lockvariable an der PORTLIST gesteuert. Durch Listenfunktionen der Form SEEK-&-SEIZE oder SEEK-&-RELEASE kann dieser REFCOUNT entsprechend manipuliert werden. Ein Löschen durch eine Funktion SEEK-&-DELETE wird nur bei REFCOUNT = 0 durchgeführt. REFCOUNT zählt dann alle Verbindungen zu einem Port und die Anzahl gerade an ihm arbeitender Routinen. Ähnliche Mechanismen braucht man auch in den LU-Areas.

Für Lock-Variablen soll neben den bekannten Funktionen LOCK und UNLOCK noch eine Funktion TESTLOCK eingeführt werden. TESTLOCK verhält sich bei freier Lockvariable wie LOCK. Ist die Variable aber belegt, wird nicht gewartet, sondern dem Aufrufer eine entsprechende Anzeige zurückgeliefert. Diese Funktion wird immer dann verwendet, wenn ein Datenbereich gesperrt werden soll und anhand der Vorgeschichte eigentlich frei sein müßte. Diese Funktion kann dadurch in einfacher Weise beitragen, Synchronisationsfehler zu erkennen und Deadlocks zu vermeiden.

Ebenfalls zur Erhöhung der Sicherheit sollte eine Lock-Variable den besetzenden Prozeß und Prozessor enthalten, und wartende Prozesse eine Zeitüberwachung durchführen (in einfacher Weise etwa Wartezyklen zählen). Ein eine Lockvariable belegender Prozeß muß befähigt werden, die Daten möglichst schnell wieder freizugeben (z.B. hohe Priorität, 'Entzugssperre'). Bei der Implementierung des Typs PORT darf eine Lock-Variable nur von den zum gleichen Rechnermodul gehörigen Prozessoren und in Sonderfällen vom PORT-Rechner bearbeitet werden. Der PORT-Rechner arbeitet dabei aber nur mit der Funktion TESTLOCK, so daß er nicht durch einen fehllaufenden lokalen Rechnermodul blockiert werden kann.

Auch bei Semaphoren wurde obiges Prinzip eingehalten. Prozesse blockieren sich nur an Semaphoren, die im eigenen Speicher liegen (genauer, vom eigenen Betriebssystem verwaltet werden). Auf Semaphoren in fremden Speichern darf nur die Funktion SIGNAL durchgeführt werden.

Die Funktion SIGNAL wird hier so realisiert, daß der aufrufende Prozeß (Prozessor) einen 'Interrupt' an den betroffenen Prozessor sendet mit einem Verweis (Capability) auf den Semaphore. Dadurch braucht der Aufrufer keine Kenntnisse über die Implementierung der Semaphore, des Prozeßbegriffes und die Abarbeitungsstrategien für Warteschlangen des angesprochenen Betriebssystems.

5.3 Globale und lokale Funktionen

Funktionen die auf einem 'lokalen' Rechnermodul ablaufen, sollen als lokale Funktionen bezeichnet werden, solche, auf dem Portrechner, als globale Funktionen. Globale Ports, d.h. in der Liste des Portrechners enthalten, können nur mit globalen Funktionen manipuliert werden.

Zunächst laufen alle in Kap.4 spezifizierten Funktionen in den Rechnermoduln als 'lokale' Funktionen los. Bei einer Anzahl von internen Funktionen wird dann erst geprüft, ob es sich um einen 'G' oder 'L' Port handelt und die entsprechende G oder L-Funktion aufgerufen. Beide Funktionen sind gleich spezifiziert, so daß im folgenden Implementierungsvorschlag nur jeweils G/L-Funktion steht

Beim Aufruf einer G Funktion stößt der Modulrechner die Funktion im Portrechner an, und wartet bis sie beendet ist. Das Warten kann über spezielle Hardwaremechanismen oder durch Setzen und Abfragen fest vereinbarter Speicherzellen realisiert werden.

G/L-Funktionen greifen nur auf PORTs und LUs zu. Der lokale Teil einer Funktion bereitet deshalb die LU soweit wie möglich vor und führt notwendige Prüfungen durch. Warten an Semaphoren darf nur in lokalen Teilen durchgeführt werden.

Die Frage, wann ein PORT lokal oder global ist wird in Anlehnung an aus der Betriebssystempraxis bekannte Verfahren gelöst. Bei CREATEPORT wird angegeben, ob der PORT lokal oder global sein soll. Bei allen anderen Funktionen wird zunächst in der lokalen Liste gesucht und bei Mißerfolg automatisch in der globalen Liste weitergesucht.

5.4 Datenbeschreibung

5.4.1 PORTLIST

TYPE	- Typkennung
LISTLOCK	- Lockvariable für Portlist
GENNAME	- vom System erzeugter Portname
LIST	- List of PORTNAME REFCOUNT PORTPOINTER
PORT:	
LOCK	- Lockvariable für Port
PORTNAME	- Name des Ports
PROCESSID	- Identifikator des Owner-Prozesses
ATTRIBUTE	- Attribute des Ports
ASSACC	- Belegmöglichkeit für Assign
LINKACC	- Belegmöglichkeit für Link
UNLINKTIME	- Zeitpunkt des Entfernens der letzten SLU
FILEPORTCOUNT	- Anzahl der angelegten Dateiports
STATISTICON	- Schalter ob Statistik Daten mitgeführt werden sollen
ORDERSUM	- Anzahl der fertiggestellten Aufträge
TIMESUM	- Summe der Verweilszeiten
QUEUELENGTH	- Anzahl der wartenden Aufträge
BEGINTIME	- Beginn der statistischen Untersuchungen
LASTCHANGE	- Zeitpunkt der letzten Änderung der Anzahl der wartenden Aufträge
QUEUESUM	- Summe der Produkte aus Anzahl der wartenden Aufträge und Zeitintervall
ASSIGNNUMBER	- Anzahl der angeschlossenen ULUs
ASSLISTPTR	- Zeiger auf ASSIGNLIST
LASS	- Länge der ASSIGNLIST
LINKNUMBER	- Anzahl der angeschlossenen SLUs
LASTACCEPT	- Index der SLU, die letzten Auftrag abholte
LINKLISTPTR	- Zeiger auf LINKLIST
LLINK	- Länge der LINKLIST
ORDERCOUNT	- Anzahl der Aufträge in der ORDERLIST
LASTIN	- Index des zuletzt eingetragenen Auftrags
NEXTOUT	- Index des Auftrags, der als nächster abgeholt werden kann
ACCLISTPTR	- Zeiger auf ACCESSLIST
LACC	- Länge der ACCESSLIST

ASSIGNLIST : array of ASGNELEMENT
 - Liste der angeschlossenen ULUs
 LINKLIST : array of LINKELEMENT
 - Liste der angeschlossenen SLUs
 ORDERLIST : equals ASSIGNLIST
 - Auftragsliste
 ACCLIST : array of IDENTIFIER
 - Liste der zugriffsberechtigten Prozesse

ASGNELEMENT/ORDERELEMENT

PROCESSID - Identifikator des Auftraggeberprozesses
 LUNR - LU-Nummer
 ULUPTR - Zeiger auf ULU

 ORDERSTATE - Auftragsstatus
 CHAININDEX - Verkettung für Auftragswarteschlange
 (Nachfolger)
 LINKINDEX - INDEX auf ausführendes LINKELEMENT

LINKELEMENT

PROCESSID - Identifikator des Serviceprozesses
 LUNR - LU-Nummer
 SLUPTR - Zeiger auf SLU

 READystate - Status der SLU (z.B. wartend auf Auftrag)
 TIMEOUT - spätestes Ende des laufenden Auftrags
 ASSINDEX - Index auf auftraggebendes ASSIGNELEMENT

5.4.2 LUAREA

LUAREA

PROCESSID	- Identifikator des Prozesses
NEXTBLPTR	- Zeiger auf Erweiterungsblock
REFCOUNT	- Referenz-Count für LU Block
ANZLU	- Anzahl der LUs im LUBODY
LUBLOCK	- array of (1..ANZLU) of LU (LU = ULU/SLU)

ULU

TYPE	- Typkennung 'ULU'
LOCK	- Lockvariable für ULU
LUNR	- Nummer der LU
PORTNAME	- Name des Ports, an den die ULU assignt ist
ATTRIBUTE	- Gewünschte Attribute
ACCMODE	- Belegmodus
ORDERSTATE	- Bearbeitungszustand des Auftrags
LUSEM	- Semaphore für Warten auf Ende
ORDERBEGIN	- Zeitpunkt des Auftrageingangs
TIMEOUT	- Späteste Endezeit
STATUS	- Replystatus
FCCODE	- Funktionscode
BUFCAP	: CAPABILITY
	- pufferbeschreibende Daten
PARPOINT	- Zeiger auf Parameterblock
PORTPOINTER	- Zeiger auf Port
ASSINDEX	- Index des Assignelementes im Port

SLU

TYPE	- Typkennung 'SLU'
LOCK	- Lockvariable für SLU
LUNR	- Nummer der LU
PORTNAME	- Name des Ports, an den SLU gelinkt ist
ATTRIBUTE	- Attribute des Serviceprozesses
ACCMODE	- Belegmodus
ORDERSTATE	- Bearbeitungszustand des Auftrags
LUSEM	- Auftragssemaphore
ORDERBEGIN	- Zeitpunkt der Bereitschaft zur Auftragsübernahme
MAXJOBTIME	- Maximale Bedienzeit
STATUS	- Replystatus
FCCODE	- Funktionscode
BUFCAP	: CAPABILITY
	- pufferbeschreibende Daten
PARPOINT	- Zeiger auf Parameterblock
PORTPOIN	- Zeiger auf Port
LINKIND	- Index des Linkelementes im Port

5.4.3 Parameterblock

PARBL

FCCODE	- Typkennung und Funktionscode
PORTNAME	- Name des Ports
LU	- Logical Unit
STATUS	- Replystatus
BUFCAP	- pufferbeschreibende Daten

5.4.4 Im Typ PORT verwendete Datenstrukturen

IDENTIFIER	= PROCESSOR: PROCESS:
PROCESSOR	- Nummer des Prozessors
PROCESS	- Name des Prozesses
CAPABILITY	- BUFPOINT: .. . BUFLENGTH: integer ACCESSRIGHTS:
BUFPOINT	- Zeiger auf Puffer
BUFLENGTH	- Pufferlänge
ACCESSRIGHTS	- Zugriffsrechte
FUNCTIONCODE	= TYPE: CODE: array (0..7) of boolean
TYPE	- Typkennung
CODE	- Funktionscode

5.5 Detailspezifikation der wichtigsten Funktionen

5.5.1 CREATEPORT (PROCESSID,PORTNAME,ATTRIBUTE,ASSACC, LINKACC,LASS,LLINK,LACC,ACCLIST,ERRVEK)

```
CHECK PARAMETER
CHECKPORTNAME(V-Name,Dt-Name GLOBLOC)
```

```
G/L - CREATEPORT:
  INSERT-NAME (PORTLIST,PORTNAME)
  if 'schon vorhanden' then goto ERROR6
  GETSTORAGE(Länge,PORTPTR)
  if 'nicht genügend Speicher' then goto ERROR7
  if Dt-Name <> nil then begin;
    CALL FILEPORTCOUNT (V-Name,+1,ERRV)
    if ERRV <> 'OK' then goto ERROR1
  end;

  PORT.(PROCESSID,PORTNAME,ASSACC,LINKACC,
    LASS,LLINK,LACC) := PARAMETER.*
  PORT.(ASSIGNNUMBER,LINKNUMBER,ORDERCOUNT) = 0
  PORT.(ASSLISTPTR,LINKLISTPTR,ACCLISTPTR) = f(PARAMETER)
  PORT.TYPE := G/L - Volume / Datei

  ACCLIST elof PORT.ACCLIST
  INSERT-VALUE (PORTLIST,PORTNAME,PORTPTR)
  goto ENDE

ERROR1 RELEASESTORAGE (Länge,PORTPTR)
ERROR7: DELETE-NAME (PORTLIST,PORTNAME)
ERROR6:

G/L - end;

ENDE end;
```

5.5.1.1 FILEPORTCOUNT (V-Name,COUNT,ERRVEK)

```

      /* COUNT= +1 wenn Fileport erzeugt wird */
      /* COUNT= -1 wenn Fileport gelöscht wird */

      SEEK & SEIZE (PORTLIST,PORTNAME)
      if 'nicht gefunden' then goto ERROR1
      if PORT.TYPE <> 'V-Port' then goto ERROR4

      LOCK (PORT.LOCK)
      if COUNT = -1 and PORT.FILEPORTCOUNT=0 then goto ERROR 11
      PORT.FILEPORTCOUNT +/-COUNT
      UNLOCK (PORT.LOCK)
      if COUNT = -1 then begin;
          SEEK & RELEASE (PORTLIST,PORTNAME)
          SEEK & RELEASE (PORTLIST,PORTNAME)
      end;
      goto ENDE;

ERROR11: UNLOCK (PORT.LOCK)
ERROR4 : SEEK & RELEASE (PORTLIST,PORTNAME)
ERROR1 :

ENDE   : end;

```

5.5.2 DELETEPORT (PROCESSID,PORTNAME,ERRVEK)

```
DELETEPORTNAME(V-Name,Dt-Name,GLOBLOC)
```

```
G/L - DELETEPORT:
```

```
PORTPTR:=SEEK & DELETE(PORTLIST,PORTNAME,ERRVEK)
if 'nicht gefunden' then goto ERROR1
if 'REFCOUNT>0' then goto ERROR3
```

```
CHECK(ASSNUMBER,LINKNUMBER,FILEPORTCOUNT,ORDERCOUNT)
if 'inkonsistent' then goto ERROR11
```

```
FREESTORAGE(PORTPTR,Länge)
if Dt Name <> nil then begin;
    CALL FILEPORTCOUNT(V-Name -1,ERRVEK)
    if ERRVEK <> 'OK' then goto ERROR
end;
```

```
goto ENDE;
```

```
ERROR11:
```

```
ERROR3
```

```
ERROR1 :
```

```
ENDE : end;
```

5.5.3 ASSIGN (PROCESSID,LUNR,PORTNAME,ACCMODE,ULUATTR ERRVEK)

GETLUAREAPTR & LOCK

```

if LUNR = 'requ' then 'suche freie LU'
if 'keine freie LU' then goto ERROR2
ULUPTR := f(LUNR,LUAREAPTR)

TESTLOCK(LU.LOCK)
PARAMETER Vorbelegung
CHECK (PRE1,PRE2)
CHECKPORTNAME (V-Name,Dt Name,GLOBLOC,ERRVEK)
ULU.(PORTNAME,ATTRIBUTE,ACCMODE) := PARAMETER.*

CALL G/L-ASSIGN (PROCESSID,ULUPTR,DTPORTNEU,ERR)
if ERR <> 'OK' then goto ERROR1

if DTPORTNEU = 'true' then begin;
  ULU.FCCODE := 'Dateiport anlegen'
  ULU.BUFCAP := Cap of 'ULU.PORTNAME-ACCMODE'
  CALL G/L-DEPOSIT2 (ULUPTR)
  CALL SEMWAIT (ULU.LUSEM)
  ULU.(FCCODE,BUFCAP) := nil
  CALL G/L-UNASSIGN (ULUPTR)
  if ULU.STATUS <> 'OK' then goto ERRORX
  CALL G/L-ASSIGN (PROCESSID,ULUPTR,DTPORTNEU,ERR)
end;

LUBLOCK.REFCOUNT += 1;
goto ASS10

ERRORX: ERRVEK := ULU.STATUS
ERROR6.
ERROR1:
ASS10 : UNLOCK (LU.LOCK)
ERROR2: UNLOCK (LUAREA.LOCK)

```

5.5.3.1 G/L-ASSIGN (PROCESSID,ULUPTR,DTPORTNEU,ERR)

```

PORTPTR := SEEK&SEIZE (PORTLIST,PORTNAME)

if 'nicht gefunden' then begin;
  PORTPTR := SEEK&SEIZE (PORTLIST,V-Name)
  if 'nicht gefunden' then goto ERROR1;
  DTPORTNEU := 'true'
end;
else DTPORTNEU := 'false'

LOCK (PORT.LOCK)
if DTPORTNEU = 'false' then begin;
  CHECK (ATTRIBUTE,ACCMODE,ACCESS)
  if 'unzulässig' then goto ERROR5/9/10;
end;

ASSELEMENTPTR:= Get-free-ASSEL (PORT.ASSLISTPTR)
if 'kein Listenplatz frei' then goto ERROR7;
ASSELEMENT.(PROCESSID,LUNR,ULUPTR) := *.*
ASSIGNNUMBER += 1;
ULU.(PORTPOINTER,ASSINDEX) := dto

ERROR5/9/10:
ERROR7:

UNLOCK (PORT.LOCK)
if ERROR then SEEK&RELEASE (PORTLIST,PORT.PORTNAME)

ERROR1

end;
```

5.5.4 UNASSIGN (PROCESSID,LUNR,PORTNAME ACCESS, ULUATTR,ERRVEK)

GET & LOCK LUAREAPTR

Parameter Vorbesetzung für LUNR = 'requ'/'all'

if LUNR = 'nil' then goto ERROR2;

do LUNR = ANF to END; /* Schleife bei 'all' */

if ULU.ORDERSTATE = 'Auftrag in Bearbeitung' then goto ERROR4;

if ULU.PORTNAME <> PARAMETER.* then goto ERROR8;

TESTLOCK (ULU.LOCK)

if 'locked' then goto ERROR11;

CALL G/L-UNASSIGN (ULUPTR,DTPORTFREI,ERR)

if DTPORTFREI then begin;

ULU.FCCODE := 'Fileport löschen

ULU.BUFCAP := Cap of ULU.PORTNAME

CALL G/L-Deposit2;

CALL SEMWAIT (ULU.LUSEM)

ULU.(FCCODE,BUFCAP) := nil

CALL G/L-UNASSIGN (ULUPTR,DTPORTFREI,ERR)

end;

ULU.PORTNAME := nil

ULU.TYPE := 'LU'

UNLOCK (LU.LOCK)

LUBLOCK.REFCOUNT -= 1

UNLOCK (LUAREA.LOCK)

end;

5.5.4.1 G/L-UNASSIGN (ULUPTR,DTPORTFREI,ERR)

PORTPTR := SEEK&SEIZE (PORTLIST,ULU.PORTNAME)

LOCK (PORT.LOCK)

PORT.ASSELEMENT.* := nil;

ASSIGNNUMBER := 1;

if ASSIGNNUMBER = 0 and TYPE = 'Fileport' then begin;

DTPORTFREI = 'true'

ULU.PORTNAME := V-Name

CALL G/L-ASSIGN (nil,ULUPTR,true,ERR)

end;

else DTPORTFREI := 'false'

UNLOCK (PORT.LOCK)

SEEK & RELEASE (PORTLIST,PORTNAME)

SEEK & RELEASE (PORTLIST,PORTNAME)

/* 2 mal RELEASE, da bei ASSIGN nicht freigegeben wird */

5.5.5 DEPOSIT (LUNR,PARPOINT,PROCEED,ERRVEK)

```

GETLUAREA & LOCK
if ULU.TYPE = 'ULU' then
    TESTLOCK (ULU.LOCK)
else ERROR merken;
UNLOCK (LUAREA.LOCK)

CHECK PRE 1,2,3 4
if ERROR then UNLOCK (ULU.LOCK) goto ENDE;

ULU.(FCCODE,BUFCAP) := PARAMETER.*

CALL G/L-DEPOSIT (ULUPTR)

if not PROCEED then
    CALL WAITLU2 (ULUPTR,PARPOINT)

UNLOCK (ULU.LOCK)

G/L-DEPOSIT (ULUPTR)

SEEK & SEIZE (PORTLIST,PORTNAME)
LOCK (PORT.LOCK)
CHECK Konsistenz
ASSELEMENT.ORDERSTATE := ULU.ORDERSTATE
ORDER in ORDERQUEUE einhängen;

if SLUs 'waiting auf Auftrag' then begin;
    GET NEXT SLU
    CALL G/L-MOVEORDER (ASSELEMENT,LINKELEMENT)
end;

UNLOCK (PORT.LOCK)

end;

```

5.5.6 REPLY (LUNR,PARPOINT,ERRVEK)

```

GETLUAREAPTR & LOCK
TESTLOCK (SLU.LOCK)
if 'belegt' then goto ERROR10
UNLOCK (LUAREA.LOCK)

CHECK PRE1
SLU.(STATUS,BUFCAP) = PARPOINT.*

CALL G/L-REPLY (SLUPTR)

SLU.ORDERSTATE := 'Auftrag zurückgegeben'
UNLOCK (SLU.LOCK)
end;
```

5.5.6.1 G/L-REPLY (SLUPTR)

```

SEEK & SEIZE (PORTLIST,PORTNAME)
ULU.(STATUS,BUFCAP) = SLU.*

ULU.ORDERSTATE := 'Auftrag beendet'
SIGNAL (ULU.LUSEM)
lösche ORDER in ORDERQUEUE
SEEK & RELEASE (PORTLIST,PORTNAME)
SEEK & RELEASE (PORTLIST,PORTNAME)
end;
```

5.5.7 WAITLU (LUNR,ORDERLU,PARPOINT,ERRVEK)

```

GETLUAREAPTR & LOCK
TESTLOCK (LU.LOCK)
if 'belegt' then goto ERROR10;
UNLOCK (LUAREA.LOCK)

CHECK PRE1

CALL SEM.WAIT(LU.LUSEM)
ORDERLU := GET geschaltete LU          1)
if ORDERLU <> LUNR then
    TESTLOCK (ORDERLU.LOCK)
if PARPOINT = 'requ' then := ORDERLU.PARPTR;

PARPOINT.(LUNR,FCCODE,PORTNAME,BUFCAP)
           := ORDERLU.*;

ORDERLU.ORDERSTATE := 'Auftrag abgeholt ;
if ORDERLU <> LUNR then
    UNLOCK (ORDERLU.LOCK);
UNLOCK (LU.LOCK)
end;

1) Mit Hilfe der OR-Verkettung werden die beteiligten
   LUs durchsucht, bis eine LU gefunden wird, mit
   ORDERSTATE = 'Auftrag beendet' bzw. 'Auftrag einge-
   troffen'.

```

6. Behandlung ausgewählter Kapitel

6.1 Bedienprozesse

Bedienungs-Kommandos an ein Betriebssystem werden in vielen Systemen durch einen eigenen Prozeß entgegengenommen und verarbeitet. Dieser Prozeß wird Kommandointerpreter Bedienprozeß oder manchmal auch einfach Systemprozeß genannt

Bei einem, wie in der Arbeit geforderten, entflochtenen Betriebssystem für Mehrprozessoranlagen erscheint es nicht sinnvoll daß ein Bedienprozeß Kommandos in einem anderen Prozessor durchführt. Man braucht in jedem Prozessor ein eigenes Exemplar eines Bedienprozesses.

Beim "Hochfahren" des Mehrprozessorsystems muß man deshalb in jedem Prozessor einen Bedienprozeß starten. Die einzelnen Prozesse müssen sich über PORTs zu einem, u.U. hierarchisch strukturierten Bedienungssystem verbinden.

Bedienungen von außen können natürlich nur an solchen Moduln ankommen, an denen eine Bedienungsstation angeschlossen ist. Der entgegennehmende Prozeß leitet Kommandos für einen anderen Prozessor mit Hilfe des Auftragssystems an den zuständigen Bedienprozeß weiter. Diese Weitergabe kann direkt erfolgen, oder auch über zentrale Knotenprozesse.

Im den folgenden Kapiteln soll unter Bedienprozeß P-BEDIEN immer ein ganzes System von Bedienprozessen verstanden werden.

6.2 Ein/Ausgabe

In bestehenden Systemen nehmen die Ein/Ausgabe-Funktionen einen wesentlichen Teil der vom System zur Verfügung gestellten Dienstleistungen ein. Im vorliegenden Mehrprozessorsystem wird angenommen, daß Ein/Ausgaben über eigene Prozesse abgewickelt werden. Da zur Erhöhung der Flexibilität die beteiligten Prozesse die Namen der Dateien bzw. E/A-Ports nicht kennen müssen, erfolgt der Verbindungsaufbau über das in 6.1 beschriebene Bedienungssystem.

Ein Beispiel soll den Gesamtablauf für folgendes Problem zeigen. Ein neues E/A-Gerät mit zugehörigem Prozeß P-GERÄT soll in ein laufendes System eingebaut und einem Anwenderprozeß P-ANWENDER verfügbar gemacht werden. Der Ablauf wird durch einen Bedienprozeß P-BEDIEN gesteuert.

Die einzelnen Prozesse führen folgende Funktionen durch.

P-BEDIEN:

* * * * *

Laden P-GERAET

Laden P-ANWENDER

(Laden soll CREATELU enthalten)

CREATEPORT (...Portname = GERAET)

LINK (P-GERAET, SLUnr, GERAET)

ASSIGN (P-ANWENDER, ULUnr, GERAET)

Starte P-GERAET (über Parameter physikalische Geraete-
adresse mitgeben)

Starte P-ANWENDER

nach Beendigung der beiden Prozesse u.U.

UNLINK ('all')

UNASSIGN ('all')

DELETEPORT (...GERAET...)

P-GERAET:

* * * * *

while "nicht Ende" do;

ACCEPT (SLUnr, Parameterblock)

WAITLU (SLUnr)

Auftrag bearbeiten

(ggf. Daten mit Hilfe des Verknuepfungsnetzes zwischen
eigenem Speicher und uebergebenem Puffer transferieren)

REPLY (SLUnr)

end while;

P-ANWENDER:

* * * * *

DEPOSIT (ULUnr, "Open"-Auftrag)

WAITLU

while "nicht Ende" do;

DEPOSIT (ULUnr, E/A-Auftrag)

WAITLU

Bearbeitungsphase

end while;

DEPOSIT (ULUnr, "Close"-Auftrag)

WAITLU

6.3 Allgemeine Dienstleistungen und Makro-Pipelining

Allgemeine Dienstleistungen, die durch eigene Prozesse u.U. auf speziellen Prozessoren ablaufen, werden vom Auftragssystem genauso behandelt wie E/A-Aufträge. Auf diesem Weg läßt sich auch ein Makro-Pipelining mit mehreren beteiligten Prozessen realisieren.

Im folgenden Beispiel sei eine Aufgabe in Teilschritte T_i zerlegbar. Jede Teilaufgabe soll durch einen eigenen Prozeß P_i bearbeitet werden.

Zur Weitergabe der Teilergebnisse zwischen den Prozessen braucht man je einen Port $PORT_i$. (Bild 6.1)

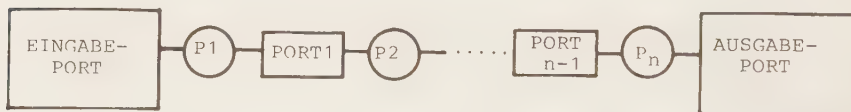


Bild 6.1:

Die beteiligten Prozesse haben dann folgenden Ablauf:

P-BEDIEN:

* * * * *

Laden aller P_i

CREATEPORT (alle $PORT_i$)

für $i = 2$ bis n :

LINK (P_i , SLUeingabe, $PORT_{i-1}$)

für $i = 1$ bis $n - 1$:

ASSIGN (P_i , ULUausgabe, $PORT_i$)

ASSIGN (P_1 , ULUeingabe, Eingabe-Port)

ASSIGN (P_n , ULUausgabe, Ausgabe-Port)

Starte alle P_i

```

Bearbeitungsprozeß Pi      * * * * *

while "nicht beendet" do;

  ACCEPT (SLUeingabe)      /* bei Pi DEPOSITE */
  WAITLU (SLUeingabe)

      Bearbeitung

  WAITLU (ULUausgabe)
      Puffer umschalten
  DEPOSITE (ULUausgabe)

end while;

```

Stellt sich beim Betrieb dieses Programmpaketes heraus, daß eine Teilaufgabe T_j längere Bearbeitungszeiten als die übrigen erfordert, d.h. einen Engpaß darstellt, so kann man dies sehr leicht beheben.

Man installiert einen zusätzlichen Prozeß P_j', der dasselbe Programm wie P_j abarbeitet, aber auf einem anderen Prozessor läuft. Diesen Prozeß P_j hängt man dann nach Bild 6.2 zusätzlich zwischen die Ports j-1 und j.

Solange Überholvorgänge bei den Aufträgen unproblematisch sind, kann ein solches Pipeliningsystem sehr leicht angepaßt werden.

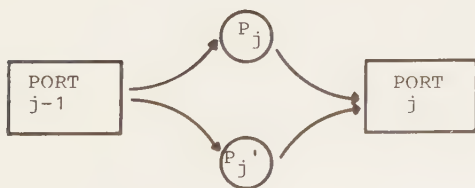


Bild 6.2:

6.4 Nachrichtenaustausch

Das Auftragssystem wurde hauptsächlich für die Vermittlung von E/A Aufträgen und anderen Dienstleistungen entworfen. Im Bedarfsfall läßt sich aber auch ein reiner Nachrichtenaustausch realisieren. Hier muß allerdings ein gewisser Overhead für Auf- und Abbau der Verbindungen in Kauf genommen werden.

Für jeden gewünschten Nachrichtenweg muß ein Port erzeugt werden. Will man in beide Richtungen "gleichzeitig" Nachrichten senden können, so müssen dafür auch zwei "Auftragskanäle" eingerichtet werden.

Es wäre auch denkbar, zwischen einer Menge von Prozessen ein Nachrichtensystem aufzubauen. Hier könnten Hilfsprozesse die Nachrichtenvermittlung steuern (ein zentraler Prozeß oder auch Ring- oder beliebige andere Netze). In einem solchen System sind dann auch Funktionen wie "Nachrichten an alle" (Broadcasting) oder Gruppenselektierung durchführbar.

Ein Nachrichten-PORT kann einen beliebigen Namen haben, den man z.B. über die Funktion GETNEWPORTNAME erzeugen kann. Das Löschen solcher PORTs wird durch den Parameter PORTNAME='all' bei der Funktion DELETEPORT erleichtert.

Der Auf- und Abbau eines Nachrichtenkanals zwischen Pi und Pk hat dann etwa folgenden Ablauf:

```
P-BEDIEN                                * * * * *

      /* Verbindungsaufbau */
GETNEWPORTNAME (P-NAME)
CREATEPORT (...PORTNAME=P-NAME...)
LINK (Pk, SLU, P-NAME)
ASSIGN (Pi, ULU, P-NAME)
.
      /* Verbindungsabbau */
UNASSIGN
UNLINK
DELETEPORT (PK, PORTNAME='all'...)
```

6.5 Synchronisation

Nimmt man den vorhandenen Overhead in Kauf, so kann man auch nur die Synchronisationsfunktionen des Auftragssystems benutzen, braucht dafür also nicht unbedingt zusätzliche, dem Anwender zugängliche, Betriebssystemdienste.

Komplexere Synchronisationsprobleme lassen sich auch hier durch Hilfsprozesse lösen, die entsprechende "Zustandsvariablen" verwalten.

Das bekannte Philosophenproblem etwa könnte durch einen zentralen Hilfsprozeß gelöst werden. Dieser Hilfsprozeß verwaltet die Zustandsvariablen (Betriebsmittel). Die einzelnen Philosophen-Prozesse melden Zustandsänderungen (Rückgabe von Betriebsmitteln) und Anforderungen über je einen Auftragskanal an den Hilfsprozeß. Bei Anforderung wartet der "Philosoph", bis ihm durch "Auftragsende" gemeldet wird, daß seine Betriebsmittel verfügbar sind.

SCHLUSSBEMERKUNG:

An dieser Stelle möchte ich mich bei Herrn Prof. Dr. F. Hofmann für die Betreuung der Arbeit und Herrn Prof. Dr. W. Händler für die Übernahme des Zweitgutachtens bedanken.

Mein Dank gilt auch Frl E. Roth für die sorgfältige Reinschrift der Arbeit und die Anfertigung der Zeichnungen.

Weiter soll allen Kollegen am Lehrstuhl für Betriebssysteme gedankt werden, die durch Diskussionen Anregungen zu dieser Arbeit geliefert haben sowie den Herren Dipl.Inf. A. Hümmer und W. Hartmann, die durch ihre Studien- und Diplomarbeiten fruchtbare Beiträge zu dieser Arbeit geleistet haben.

LITERATURVERZEICHNIS:

- /BalR71/ Balzer, R.M.
 PORTS - A Method for dynamic Interprogramm Communication
 and Job Control
 AFIPS Conf. Proc. (May 1971)

- /BocG79/ Bochmann v. G.:
 Architecture of Distributed Computer Systems
 Lecture Notes in Computer Science
 Springer-Verlag, Berlin/Heidelberg/New York (1979)

- /BöhK79/ Böhm, K.; Peter, G..
 Process Communication for Distributed Systems
 S.Schindler (ed.) Kommunikation in verteilten Systemen
 Workshop der GI in Berlin 1979
 Springer-Verlag 1979, pp 296-316

- /BriP75/ Brinch Hansen, P.:
 The Programming Language CONCURRENT PASCAL
 IEEE Transactions on Software Engineering, Vol. Se-1,
 No.2, June 1975, pp 199-207

- /CamR74/ Campbell, R.H.; Habermann, A.N.:
 The Specification of Process Synchronization by Path Ex-
 pressions
 Lecture Notes in Comp. Sc., Vol.16
 Springer-Verlag (1974)

- /CasL77/ Casey, L.; Shelness, N.:
 A Domain Structure for Distributed Computer Systems
 ACM Proceedings of Sixth Symposium on Operating Systems
 Principals 1977

- /DenP76/ Denning, P.J.:
 Fault Tolerant Operating Systems
 ACM Computing Survey, Vol.8, No.4 1976, p 359-387

- /DetJ72/ Detreville, J.:
 Designe and Implementation of an Example Operating
 System
 MIT, Thesis 1972

- /EnsP75/ Enslow, P.E.:
Multiprocessors and Other Parallel Systems an
Introduction and Overview
W. Händler (ed.), Computer Architecture, Workshop of the
GI in Erlangen 1975
Springer-Verlag 1976, pp 133-198
- /EnsP77/ Enslow, P.E.:
Multiprocessor Organization - A Survey
ACM Computing Survey, Vol. 9, No.1, 1977, pp 61 102
- /GerB76/ Gernoth, B.; Schmidt, B.:
Das Modell MEM-SIM. Beschreibung eines Simulations-
modells zur Untersuchung einer Rechenanlage mit einer
symmetrischen Mehrprozessorkonfiguration und privaten
Speichern.
Arbeitsberichte des IMMD, Univ. Erlangen-Nürnberg, Band
9 (1976)/5
- /HabA76/ Habermann, A.N.; Flow, L.; Coopride, L.:
Modularisation and Hierarchy in a Family of Operating
Systems
CACM, Vol.19, No.5, 1976, pp 266-272
- /HänW75/ Händler, W.; Hofmann, F.; Schneider, H.J.:
A General Purpose Array with a Broad Spectrum of Appli-
cations.
Informatik-Fachberichte 4, Computer Architecture,
Workshop of the GI in Erlangen 1975
Springer-Verlag 1976
- /HänW77/ Händler, W.:
Aspects of Parallelism in Computer Architecture
IMACS (AICA)-GI-Symposium
"Parallel Computers - Parallel Mathematics", München,
März 1977
- /JäpD77/ Jäpel, D.:
Ein Begriffssystem zur Formulierung von Schutzaspekten
und seine Anwendung auf Programmsysteme
Diplomarbeit, IMMD-Arbeitsbericht, FAU Erlangen-Nürn-
berg, Band 10, Nr.7, 1977
- /JonA77/ Jones, A.K.:
The Object Model A Tool for Structuring Software
Department of Computer Science Carnegie-Mellon Uni-
versity Pittsburgh, July 1977

- /KerS74/ Keramidis, S.:
Ein Beitrag zur Theorie der Prozesssysteme
Arbeitsbericht des IMMD, Univ. Erlangen-Nürnberg,
Band 9 (1974)/3
- /KerS74a/ Keramidis, S.; Bolch, G.:
Betriebsprogrammierung I
Skripten einer Vorlesung von Prof. Dr. F. Hofmann,
Erlangen, 1974
- /KobR76/ Kober, R.; Kopp, H.; Kuzina, Ch.:
SMS 101 - A Structured Multimicroprocessor-System with
Deadlock Free Operation Scheme
Euromicro Newsletter, 2(1976), pp 56-64
- /LisB75/ Liskov, B.; Zilles, S.:
Specification Techniques for Data Abstractions
IEEE Trans. on Software Engineering, 1.1 (March 1975),
pp 7-18
- /MetR76/ Metcalfe, R.M.; Boggs, D.R.:
Ethernet: Distributed Packet-Switching for Local Com-
puter Networks
CACM Vol.7 (1976), pp 395-404
- /NehJ72/ Nehmer, J.:
Ein PL/1-Programm zur Simulation von Dispatching-Strate-
gien in einem Multiprozessorsystem
In: Rechner- und Betriebssystem-Analyse, Simulation und
Entwurf, Nachrichtentechnische Fachberichte, Band 44,
VDE-Verlag Berlin, 1972, pp 170-178
- /OrnS75/ Ornstein, S.M.; et al.:
Pluribus - A Reliable Multiprocessor
National Computer Conference, 1975, AFIPS Press
pp 551-559
- /ParD72/ Parnas, D.L.:
A Technique for Software Module Specification with Ex-
amples, CACM 15, 5(1972), pp 330-336
- /ParD72a/ Parnas, D.L.:
On the Criteria to be used in Decomposing Systems into
Modules
CACM 18, 7(1975), pp 401-408
- /ParD76/ Parnas, D.L.; Würges, H.:

Response to Undesired Events, TH Darmstadt 1976

- /RudJ72/ Rudolph, J.A.:
A Production Implementation of an Associative Array
Processor - Staran
FJCC 1972, pp 229-241
- /StoH75/ Stone, H.S.; Chen, T.C.; Flynn, M.J.; Fuller, S.H.;
Lane, W.G.; Loomis, H.H.; McKeeman, W.M.; Magleby, K.B.;
Matick, R.E.; Whitney, T.M.:
Introduction to Computer Architecture
SRA 1975
- /SwaR77/ Swan, R.J. et al:
Cm* - A Modular Multi-Microprocessor
AFIPS-Conf. Proc. Vol.46, pp 637-644
- /ThoJ70/ Thornton, J.E.:
Design of a Computer: The Control Data 6600
Foresman and Company, Glenview Illinois 1970
- /WirN71/ Wirth, N.:
The Programming Language PASCAL
Acta Informatica 1, 1971, pp 35-63
- /WitF75/ Withington, F.G.:
Beyond 1984: A Technology Forecast
Datamation, Jan.1975, pp 54-73
- /WulW72/ Wulf, W.A; Bell, C.G:
C.mmp - A Multi-Mini-Processor
AFIPS Conf. Proc., FJCC 1972, pp 765-777
- /WulW74/ Wulf, W.A. et al.:
HYDRA : The Kernel of a Multiprocessor Operating System
Comm. ACM 17, 6(Juni 1974), pp 337-345
- /ZimH75/ Zimmermann, H.:
The Cyclades end to end protocol
Proc. Fourth Data Communications Symposium, ACM/IEEE,
1975 pp 7.21-7.26

Lebenslauf

Franz Xaver W u r m

geb. am : 12.5.1940

in : Oberölsbach / Deutschland

Staatsang. : deutsch

Eltern : Franz Wurm (+)
Ehefrau Hedwig, geb. Kürzinger

Familienstand: verheiratet seit 1968
Ehefrau Dorle, geb. Stegmeier
zwei Kinder

Schulen : 1946-1953 Volksschule Gnadenberg
1954-1961 Hum. Gymnasium Eichstätt

Studium : 1961-1969 Physikstudium an der
Universität Erlangen-Nürnberg

Prüfungen : 1966 Diplomvorprüfung Physik
1969 Diplomhauptprüfung Physik

Stellungen : 1969-1972 Entwicklungsingenieur bei
Fa. Triumph AG, Nürnberg
1972-1974 Entwicklungsingenieur bei
Fa. Siemens AG, Erlangen
(über Fa. GEP München)
seit 1974 Wiss. Angestellter am IMMD,
Inf. IV der Univ. Erlangen-Nürnberg

Erlangen, den 16.4.1980

